I16: Version Number

Version number is the version identifier for this PMI Polygon Data Element. V9.5 format only supports version 1 of the PMI Polygon data

I32: Reserved Field

Reserved Field is a data field reserved for future JT format expansion

VecI32: vNumVerts

An integer vector used to record the number of vertices in each polygon data element. The length of this vector is equal to the number of PolygonData elements written in this block of PMI PolygonData. The presence of additional data fields in each PolygonData element is hinged upon that element having more than 0 vertices recorded in this vector.

Retrieve next vertCount from vNumVerts

If the next element in the vNumVerts vector is non-zero, proceed to read other fields that make up a single PMI PolygonData element. Otherwise, skip reading more data for this element and loop back to seek the next element in the vector.

iNumVerts

Number of vertices for the i^{th} PolygonData element.

Length Of vNumVerts

Number of Polygon Data elements.

I32: NormalBinding

A Boolean value that indicates if there are normals present along with the list of coordinates at each vertex.

I32: ColorBinding

A Boolean value that indicates if there are colors present along with the list of coordinates at each vertex.

I32: TextureBinding

A Boolean value that indicates if there are Texture Coordinates present along with the list of coordinates at each vertex.

I32: PolygonDimension

Indicates the dimension of vertex coordinates.

VecI32: PrimTypes

An array indicating the type of each of the primitive stored in the PrimIndices array. Adjacent numbers in the array form tuples of the form [PrimIndex, PrimType]. All primitives to the left of the PrimIndex are of type PrimType unless they are already to the left of an earlier PrimIndex in this array.

VecI32: PrimIndices

Indices of vertices that form a single primitive. The difference between two adjacent values in this array determines the length of the primitive. An extra element is stored at the end of this array to identify the length of the last primitive. Values in this array are indices into the VertIndices array.

VecI32: VertIndices

An array of indices into the Vertices array. This index array eliminates the need to duplicate floating point vertices that are shared by multiple primitives.

VecF32: Vertices

The list of vertex coordinates. Each vertex is made of PolygonDimension coordinates. The length of this list is equal to number of vertices multiplied by PolygonDimension.

VecF32: Normals

An optional list of Normals for each vertex. Presence of this list is indicated by the NormalBinding flag. Each normal consists of PolygonDimension components. The size of this list is equal to number of vertices multiplied by PolygonDimension.

VecF32: Colors

An optional list of Colors for each vertex. Presence of this list is indicated by the ColorBinding flag. Each color consists of PolygonDimension components. The size of this list is equal to number of vertices multiplied by PolygonDimension.

VecF32: Texture Coords

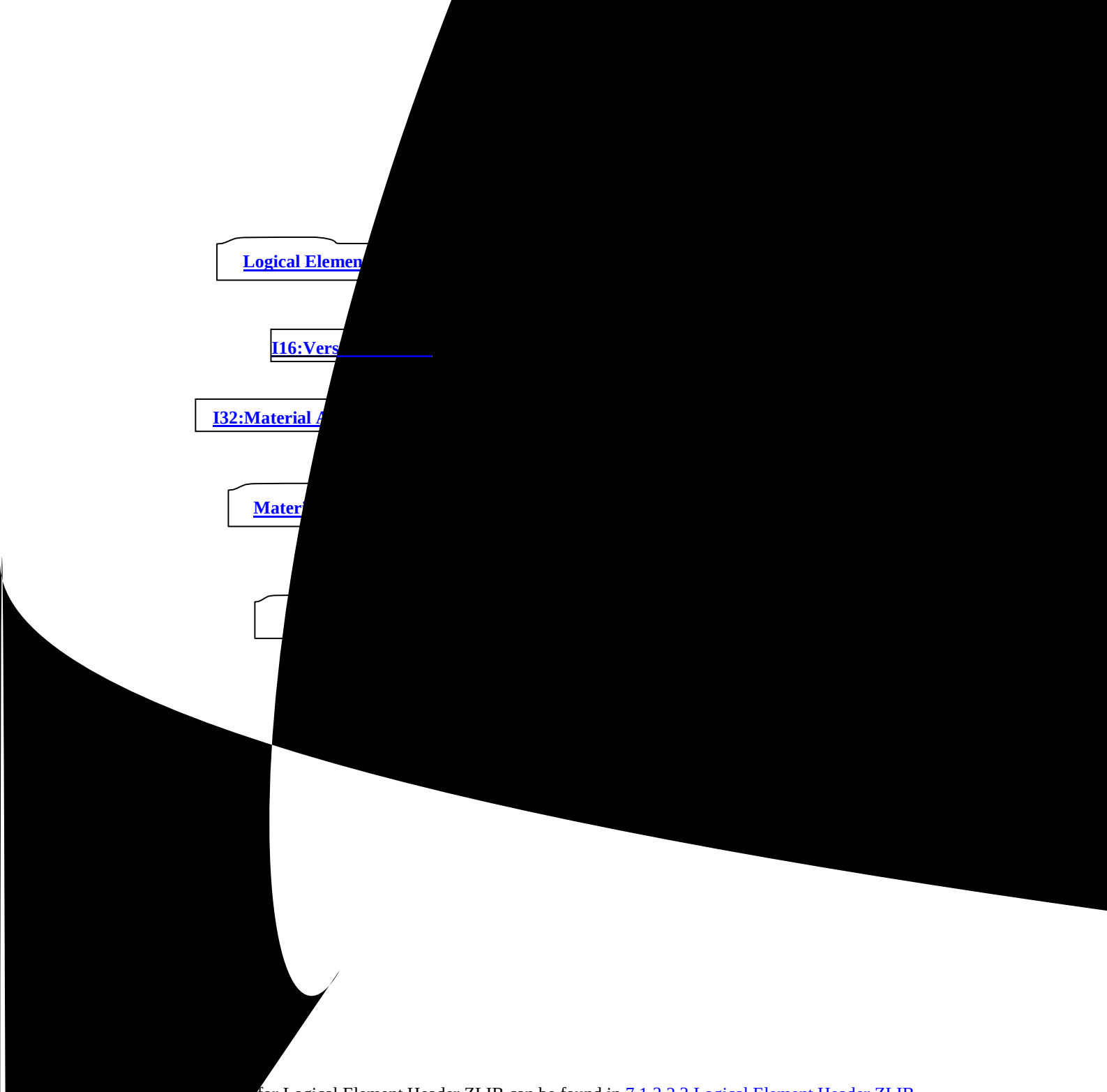
An optional list of Texture coordinates for each vertex. Presence of this list is indicated by the TexCoordBinding flag. Each TexCoord consists of 2 components. The size of this list is equal to number of vertices multiplied by 2.

7.2.7 PMI Data Segment

The PMI Manager Meta Data Element (as documented in 7.2.6.2 PMI Manager Meta Data Element) can sometimes also be represented in a PMI Data Segment. This can occur when a pre JT 8 version file is migrated to JT 9.5 version file. So from a parsing point of view a PMI Data Segment should be treated exactly the same as a 7.2.6 Meta Data Segment.

7.2.8 JT ULP Segment

JT ULP Segment contains an Element that defines the semi-precise geometric Boundary Representation data for a particular Part in JT ULP format. Note that there is also two other Boundary Representation formats (i.e. JT B-Rep and XT B-Rep) supported by the JT file format within a different file Segment Type. Complete description for the JT B-Rep and the



for Logical Element Header ZLIB can be found in 7.1.3.2.3 Logical Element Header ZLIB.

umber

r is the version identifier for this JT ULP Element 9HUVLRQ QXPEHUV ³ ´ DQG ³ ´ DUH FXUUHQWO\ VXSSRUWHG

rial Attribute Element Count

Attribute Element Count is the number of material attribute elements.

lete description for Material Attribute Element can be found in 7.2.1.1.2.2 Material Attribute Element.

Figure 174: Topological Entity Counts data collection

I32 : Region Count

Region Count indicates the number of topological region entities in the ULP.

I32 : Shell Count

Shell Count indicates the number of topological shell entities in the ULP.

I32 : Face Count

Face Count indicates the number of topological face entities in the ULP.

I32 : Loop Count

Loop Count indicates the number of topological loop entities in the ULP.

I32 : CoEdge Count

CoEdge Count indicates the number of topological coedge entities in the ULP.

I32 : Edge Count

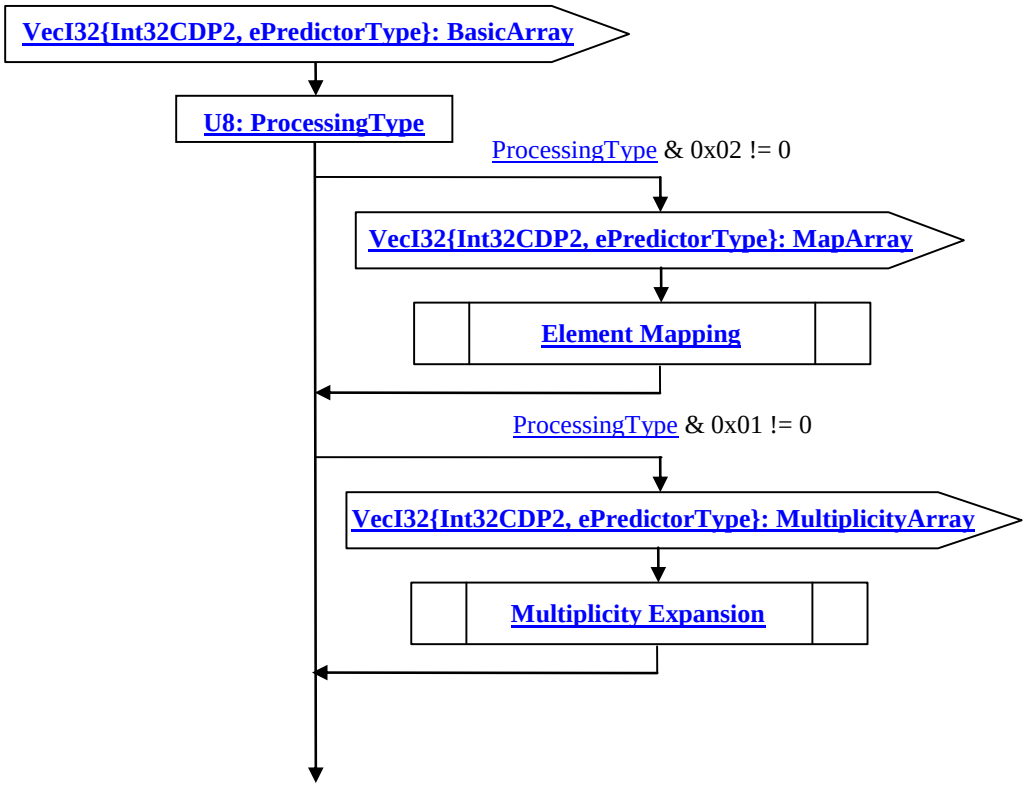
Edge Count indicates the number of topological edge entities in the ULP.

I32 : Vertex Count

Vertex Count indicates the number of topological vertex entities in the ULP.

7.2.8.1.1.2 Combined Predictor Ty

Figure 175: Combined Predictor Type data collection



VecI32{Int32CDP2, ePredictorType}: BasicArray

BasicArray is an integer array, compressed and encoded using the Int32 version of second generation CODEC.

U8: ProcessingType

Two bits of this value are currently used. If bit 0x02 is set, then the integer array is a list of elements with unique values and Element Mapping step is needed to recover the original values. If bit 0x01 is set, then the some elements in the integer array may be repeated, and Multiplicity Expansion is used to recover the original values.

VecI32{Int32CDP2, ePredictorType}: MapArray

MapArray is an integer array, where each element represents the index mapping information. MapArray is compressed and encoded using the Int32 version of second generation CODEC.

Element Mapping

Element Mapping recovers the original array from BasicArray and MapArray, using relationship $OriginalArray[i] = BasicArray[MapArray[i]]$. After Element Mapping, the value of $BasicArray$ is updated with $OriginalArray$.

VecI32{Int32CDP2, ePredictorType}: MultiplicityArray

MultiplicityArray is an integer array, where each element represents the multiplicity of each element in BasicArray. MultiplicityArray is compressed and encoded using the Int32 version of second generation CODEC.

Multiplicity Expansion

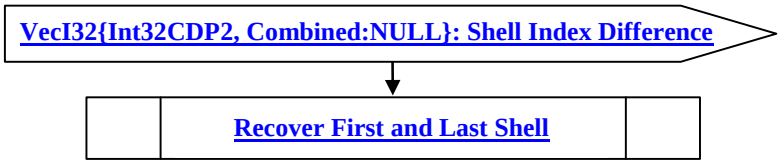
Multiplicity Expansion recovers the original array from BasicArray and MultiplicityArray. The original array is an expansion of the BasicArray. If the corresponding multiplicity value is greater than 1, the element in BasicArray is contiguously repeated in the original array according to multiplicity value.

7.2.8.1.1.3 Regions Topology Data

Regions Topology Data defines the disjoint set of non-overlapping Shells making up each Region. Each Region is defined by one or more non-overlapping **6KH00V 7KH YR0XPH RI D 5HJLRQ LV WKDW YR0XPH 0\LQJ LQVLGH HDFK ³DQWL-KR0H 6KH00´ DQG** outside each simply-**FRQWDLQHG ³KR0H 6KH00´ EH0RQJLQJ WR WKH SDUWLFX0DU 5HJLRQ $ 5HJLRQ LV DQD0RJRXV WR D GLPHQVLRQDO0** elevated face where Region corresponds to Face and Shell corresponds to Trim Loop.

(DFK 5HJLRQ¶V GHILQLQJ 6KH00V DUH LGHQWLILHG LQ D 0LVW RI 6KH00V E\ DQ LQGH[IRU ERWK WKH ILUVW 6KH00 DQG WKH 0DVW 6KH00 LQ Hach Region (i.e. all Shells inclusive between the specified first and last Shell list index define the particular Region). In addition, the indices of all the shells in a single Region are contiguous. The first shell index of the first region is 0, and the first shell index of other regions is one greater than the last shell index of the previous region. Therefore only the number of shells of each region is stored. In the special case when the number of regions is 1, no information needs be stored since its last Shell index is known to be Shell Count-1.

Figure 176: Regions Topology Data collection



VecI32{Int32CDP2, Combined:NULL}: Shell Index Difference

Shell Index Difference is a vector of indices representing the integer value by subtracting first shell index from last shell index in each region, encoded using Combined Predictor Type. Shell Index Difference is compressed and encoded using the Int32 version of second generation CODEC.

Recover First and Last Shell Indices

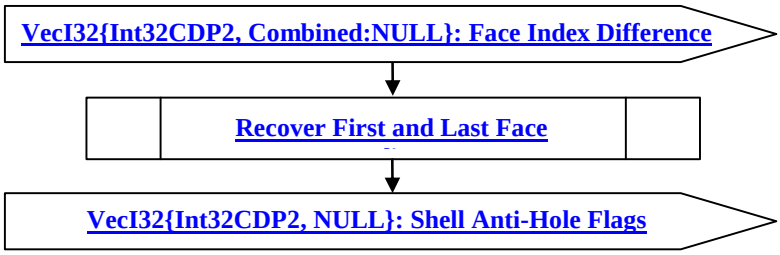
The first shell index of the first region is 0, and the last shell index of the first region is element 0 of Shell Index Difference. The first shell index of region $k, k \geq 1$ equals to the last shell index of region $k-1$ plus 1. The last shell index of region $k, k \geq 1$ equals to the first shell index of region k plus element k of Shell Index Difference array.

7.2.8.1.1.4 Shells Topology Data

Shells Topology Data **GHILQHV WKH VHW RI WRSR0RJLFD0 DGMDFHQW)DFHV PDNLQJ XS HDFK 6KH00 $ 6KH00¶V VHW RI WRSR0RJLFD0** adjacent Faces define a single (usually closed) two manifold solid that in turn defines the boundary between the finite volume of space enclosed within the Shell and the infinite volume of space outside the Shell. In addition, each Shell has a flag that **GHQRWHV ZKHWKHU WKH 6KH00 UHIHUV WR WKH ILQLWH LQWHULRU YR0XPH L H D ³KR0H 6KH00´ RU WKH LQILQLWH H[WHULRU YR0XPH L H DQ ³DQWL-KR0H 6KH00´**

(DFK 6KH00¶V defining Faces are identified in a list of Faces by an index for both the first Face and the last Face in each Shell (i.e. all Faces inclusive between the specified first and last Face list index define the particular Shell). In addition, the indices of all the faces in a single Shell are contiguous. The first face index of the first shell is 0, and the first face index of other shells is one greater than the last face index of the previous shell. Therefore only the number of faces of each shell is stored. In the special case when the number of shells is 1, no information needs be stored since its last face index is known to be Face Count-1.

Figure 177: Shells Topology Data collection



VecI32{Int32CDP2, Combined:NULL}: Face Index Difference

Face Index Difference is a vector of indices representing the integer value by subtracting first face index from last face index in each shell, encoded using Combined Predictor Type. Face Index Difference is compressed and encoded using the Int32 version of second generation CODEC.

Recover First and Last Face Indices

The first face index of the first shell is 0, and the last face index of the first shell is element 0 of Face Index Difference. The first face index of shell $k, k \geq 1$ equals to the last face index of shell $k-1$ plus 1. The last face index of shell $k, k \geq 1$ equals to the first face index of shell k plus element k of Face Index Difference array.

VecI32{Int32CDP2, NULL}: Shell Anti-Hole Flags

Each Shell has a flag identifying whether the Shell is an anti-hole Shell. Shell Anti-Hole Flags is a vector of anti-hole flags for a set of Shells.

In an uncompressed/decoded form the flag values have the following meaning:

= 0	Shell is not an anti-hole Shell
= 1	Shell is an anti-hole Shell

Shell Anti-Hole Flags uses the Int32 version of the CODEC to compress and encode data.

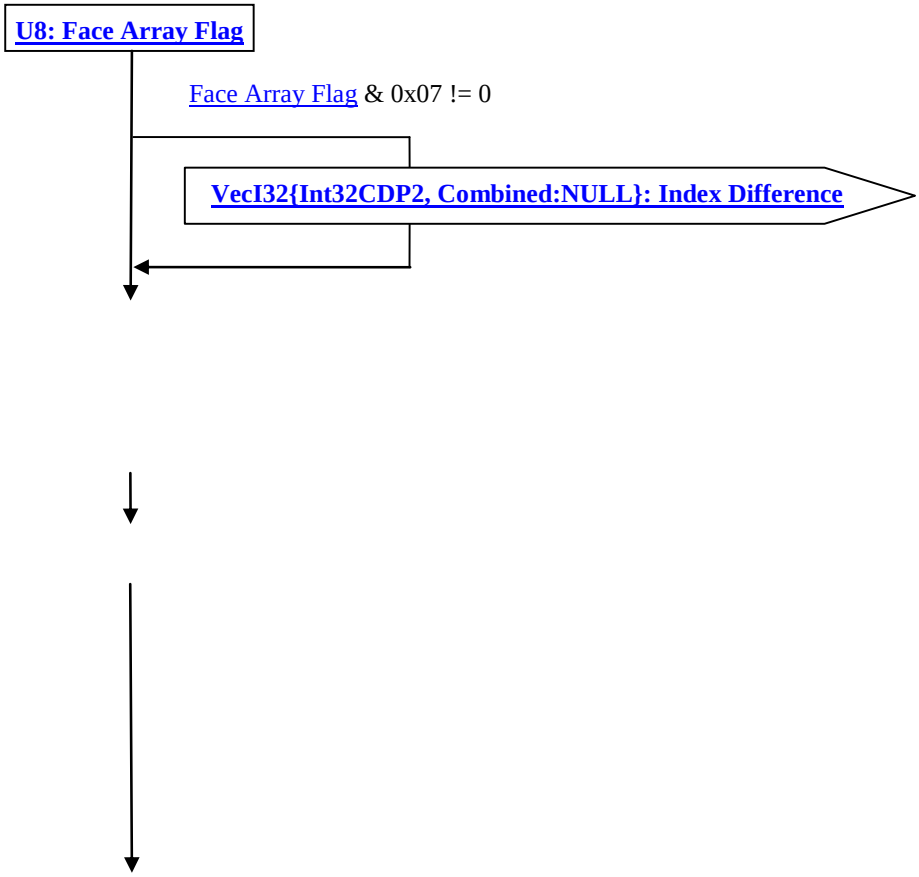
7.2.8.1.1.5Faces Topology Data

$)DFH PXVW EH WULPPHG ZLWK DW 0HDVW RQH ³DQWL-KR0H´ 7ULP /RRS DQG PD\ EH WULPPHG ZLWK RQH RU PRUH ³KR0H´ 7ULP /RRSV The complete description of face and its relation to the trim loops can be found in 7.2.3.1.3.3 Faces Topology Data.

(DFK)DFH¶V GHILQLQJ 7ULP /RRSV DUH LGHQWLfied in a list of trim Loops by an index for both the first Trim Loop and the last Trim Loop in each Face (i.e. all Trim Loops inclusive between the specified first and last Trim Loop list index define the particular Face). In addition, the indices of all the loops in a single Face are contiguous. The first loop index of the first face is 0, and the first loop index of other faces is one greater than the last loop index of the previous face. Therefore only the number of loops of each face is stored. In the special case when the number of faces is 1, no information needs be stored since its last loop index is known to be Loop Count-1.

(DFK)DFH¶V XQGHU\LQJ *HRPHWULF 6XUIDFH LV LGHQWLILHG E\ DQ LQGH[LQWR D OLVW RI *HRPHWULF 6XUIDFHV (DFK IDFH¶V PDWHULDO LV identified by an index into the list of Material Attribute Elements.

Figure 178: Faces Topology Data collection



U8: Face Array Flag

Face Array Flag indicates which arrays of face topology data are not trivial and therefore encoded.

VecI32{Int32CDP2, Combined:NULL}: Index Difference Array

Index Difference Array is a combined vector of indices encoded using Int32 version of CODEC and Combined Predictor Type, with its content decided by the value of Face Array Flag. If Face Array Flag has bit 0x01 set, then the vector of integer values obtained by subtracting first loop index from last loop index in each face is appended to the end of Index Difference Array. If Face Array Flag has bit 0x02 set, then the vector of integer values obtained by subtracting surface index from face index in each face is appended to the end of Index Difference Array. If Face Array Flag has bit 0x04 set, then the vector of integer values representing the material index of each face is appended to the end of Index Difference Array.

Recover First and Last Loop Indices

The first loop index of the first face is 0, and the last loop index of the first face is element 0 of Index Difference Array if the array is encoded, or 0 if bit 0x01 of Face Array Flag is not set. The first loop index of face $k, k \geq 1$ equals to the last loop index of face $k-1$ plus 1. The last loop index of face $k, k \geq 1$ equals to the first loop index of face k plus element k of Index Difference Array, or 0 if bit 0x01 of Face Array Flag is not set.

Recover Surface Indices

The surface index of each face equals to the face index if bit 0x02 of Face Array Flag is not set. Otherwise the surface index of face k is obtained by substracting element $k + offset$ of Index Difference Array from face index k, where $offset$ is equal to Face Count if bit 0x01 of Face Array Flag is set and 0 if the bit is not set.

Recover Material Indices

The material index of each face equals to 0 if bit 0x04 of Face Array Flag is not set. Otherwise the material index of face k equals to the element $k + offset$ of Index Difference Array, where $offset$ is equal to twice of Face Count if both bit 0x01 and bit 0x02 of Face Array Flag are set, is equal to Face Count if either bit 0x01 or bit 0x02 of Face Array Flag is set, and is equal to 0 if neither bit is set.

Vecl32{Int32CDP2, Combined:NULL}: Flag Bit Array

Only the lower 24 bits of the four integer indices, namely first loop index, last loop index, surface index, and material index, are used as integer identifiers. The other bits of these integers are either used to encode additional information, or reserved for future usage.

	24	25	26	27	28	29	30	31
First Loop Index	Surface Type			U Knot Type		V Knot Type		isNormalReversed
Last Loop Index	isIsolated	Reserved						
Surface Index	Reserved							
Material Index	Reserved							

Each element of Flag Bit Array is a 32 bit integer obtained by combining all 32 flag bits from four different integers. More specifically:

- Bits 0~7 of Flag Bit Array are equal to bits 24~31 of First Loop Index
- Bits 8~15 of Flag Bit Array are equal to bits 24~31 of Last Loop Index
- Bits 16~23 of Flag Bit Array are equal to bits 24~31 of Surface Index
- Bits 24~31 of Flag Bit Array are equal to bits 24~31 of Material Index

Supported Surface Type

In an uncompressed/decoded form, the supported surface types are listed below.

0	Nurbs
1	Plane
2	Cylinder
3	Cone
4	Sphere
5	Torus
6	Reserved
7	Reserved

Supported Knot Type

In an uncompressed/decoded form, the supported knot types are listed below. The knot type of the underlying surface along both U and V parameter directions are encoded.

0	No Pattern
1	No knot value in between the clamped end knots
2	All knot values in between the end knots increase with an even interval
3	All knot values in between the end knots repeat exactly once, and the distinct values increase with an even interval

In an uncompressed/decoded form, the Face Reverse Normal Flag has the following meaning:

= 0	Face normal is not reversed
= 1	Face normal is reversed.

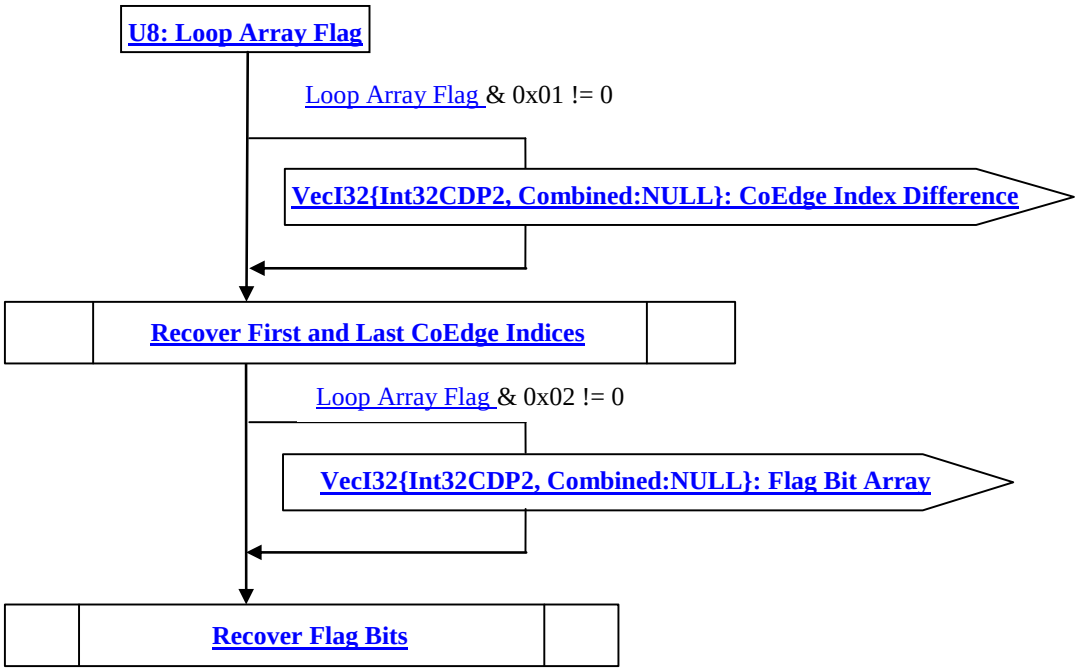
Recover Flag Bits

If Face Array Flag & 0x08 is equal to 0, then each element in Flag Bit Array is set to have value 0. The flag bits are recovered by assigning bits 0~7 of Flag Bit Array to bits 24~31 of First Loop Index, bits 8~15 of Flag Bit Array to bits 24~31 of Last Loop Index, bits 16~23 of Flag Bit Array to bits 24~31 of Surface Index, and bits 24~31 of Flag Bit Array to bits 24~31 of Material Index.

7.2.8.1.1.6 Loops Topology Data

A Loop (often called Trimming Loop) defines in parameter space a 1D boundary around which geometric surfaces are trimmed to form a Face. Loops Topology Data specifies the CoEdges making up each Loop along with an anti-hole flag and identifier tag for each Loop. The complete description of loop and its relation to the CoEdges can be found in 7.2.3.1.3.4 Loops Topology Data.

Figure 179: Loops Topology Data collection



U8: Loop Array Flag

Loop Array Flag indicates which arrays of loop topology data are not trivial and therefore encoded.

VecI32{Int32CDP2, Combined:NULL}: CoEdge Index Difference

CoEdge Index Difference is a vector of indices representing the integer value by subtracting first CoEdge index from last CoEdge index in each loop, encoded using Combined Predictor Type. CoEdge Index Difference is compressed and encoded using the Int32 version of second generation CODEC.

Recover First and Last CoEdge Indices

The first CoEdge index of the first loop is 0, and the last CoEdge index of the first loop is element 0 of CoEdge Index Difference. The first CoEdge index of loop $k, k \geq 1$ equals to the last CoEdge index of loop $k-1$ plus 1. The last CoEdge index of loop $k, k \geq 1$ equals to the first CoEdge index of loop k plus element k of CoEdge Index Difference array.

VecI32{Int32CDP2, Combined:NULL}: Flag Bit Array

Only the lower 24 bits of the two integer indices, namely first CoEdge index and last CoEdge index are used as integer identifiers. The other bits of these integers are either used to encode additional information, or reserved for future usage.

	24	25	26	27	28	29	30	31
First CoEdge Index	Reserved							isAntiHoleLoop
Last CoEdge Index	Reserved							

Bits 0~7 of Flag Bit Array are equal to bits 24~31 of First CoEdge Index

Bits 8~15 of Flag Bit Array are equal to bits 24~31 of Last CoEdge Index

Bits 16~31 of Flag Bit Array are set to be 0

In an uncompressed/decoded form, the AntiHole Loop Flag has the following meaning:

= 0	Loop is not an anti-hole Loop
= 1	Loop is an anti-hole Loop

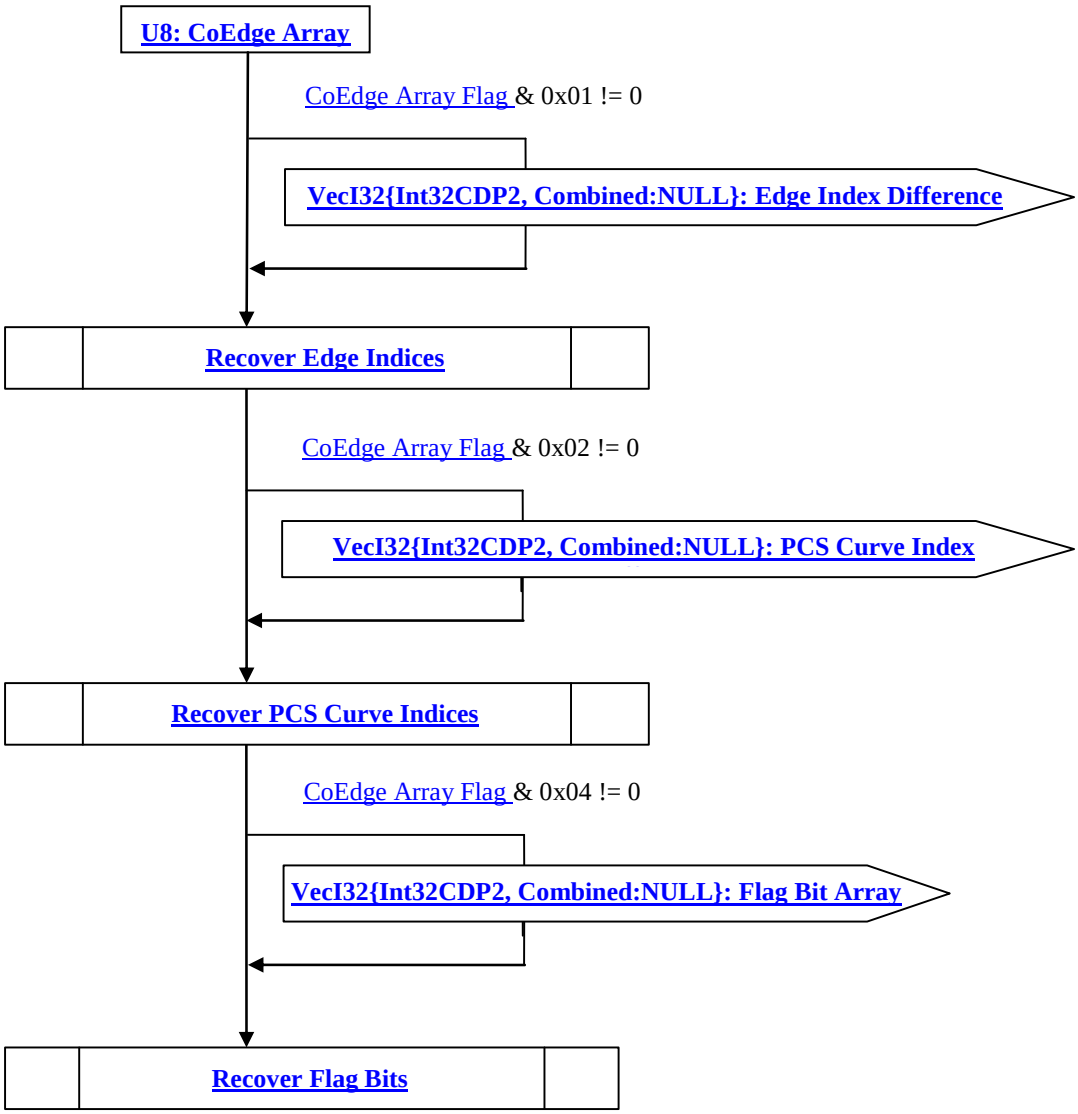
Recover Flag Bits

The flag bits are recovered by assigning bits 0~7 of Flag Bit Array to bits 24~31 of First CoEdge Index, and bits 8~15 of Flag Bit Array to bits 24~31 of Last CoEdge Index.

7.2.8.1.1.7 CoEdges Topology Data

A CoEdge defines a parameter space edge trim Loop segment (i.e. the projection of an Edge into the parameter space of the Face). CoEdges Topology Data specifies the underlying Edge and PCS Curve making up each CoEdge along with a MCS curve reversed flag and tag for each CoEdge. The complete description of CoEdge and its relation to the Edge can be found in 7.2.3.1.3.5 CoEdges Topology Data.

Figure 180: CoEdges Topology Data collection



U8: CoEdge Array Flag

CoEdge Array Flag indicates which arrays of coedge topology data are not trivial and therefore encoded.

VecI32{Int32CDP2, Combined:NULL}: Edge Index Difference

Edge Index Difference is a vector of indices representing the integer value by subtracting the Edge index from the CoEdge index for each CoEdge, encoded using Combined Predictor Type. Edge Index Difference is compressed and encoded using the Int32 version of second generation CODEC.

Recover Edge Indices

If CoEdge Array Flag & 0x01 is equal to 0, then the Edge index of each CoEdge is equal to the CoEdge index. Otherwise, the Edge index of CoEdge with index k can be computed by substracting element k of Edge Index Difference array from k, the CoEdge index.

VecI32{Int32CDP2, Combined:NULL}: PCS Curve Index Difference

PCS Curve Index Difference is a vector of indices representing the integer value by subtracting the PCS Curve index from the CoEdge index for each CoEdge, encoded using Combined Predictor Type. PCS Curve Index Difference is compressed and encoded using the Int32 version of second generation CODEC.

Recover PCS Curve Indices

If CoEdge Array Flag & 0x02 is equal to 0, then the PCS Curve index of each CoEdge is equal to the CoEdge index. Otherwise, the PCS Curve index of CoEdge with index k can be computed by substracting element k of PCS Curve Index Difference array from k, the CoEdge index.

VecI32{Int32CDP2, Combined:NULL}: Flag Bit Array

Only the lower 24 bits of the two integer indices, namely Edge index and PCS Curve index, are used as integer identifiers. The other bits of these integers are either used to encode additional information, or reserved for future usage.

	24	25	26	27	28	29	30	31
Edge Index	Knot Type			Domain Type		PCS Curve Type		isXYZReversed
PCS Curve Index	isUvInc	Reserved						

Bits 0~7 of Flag Bit Array are equal to bits 24~31 of Edge Index

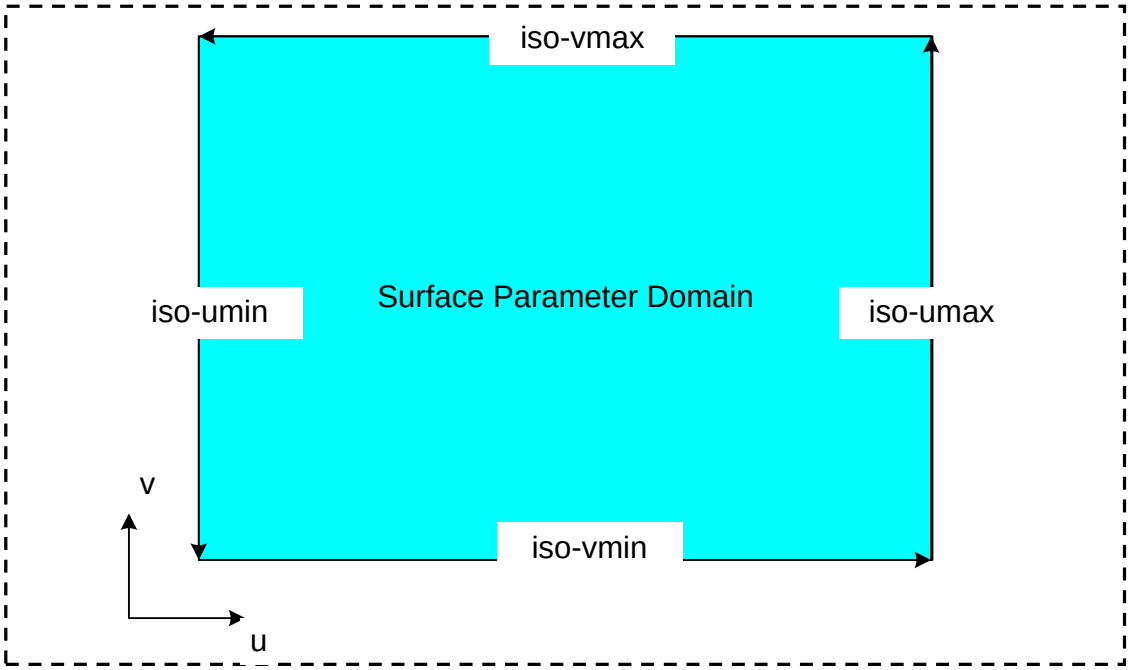
Bits 8~15 of Flag Bit Array are equal to bits 24~31 of PCS Curve Index

Bits 16~31 of Flag Bit Array are set to be 0

The Knot Type, defined in Supported Knot Type, is an integer with its value between 0 and 3.

Domain Type

Figure 181: Surface Domain Classification



In an uncompressed/decoded form, the supported PCS Curve types are listed below.

0	General
1	PCS curve is coincident with iso-umin curve in the surface parameter domain
2	PCS curve is coincident with iso-umax curve in the surface parameter domain
3	PCS curve is coincident with iso-vmin curve in the surface parameter domain
4	PCS curve is coincident with iso-vmax curve in the surface parameter domain
5	Reserved
6	Reserved
7	PCS curve is to be derived from MCS curve and surface geometry

PCS Curve Type

In an uncompressed/decoded form, the supported PCS Curve types are listed below.

0	Nurbs
1	Line
2	Circle
3	Reserved

In an uncompressed/decoded form, the XYZReversed Flag has the following meaning:

= 0	Directional sense of associated edges MCS curve should not be interpreted as opposite the direction its parameterization implies.

= 1	Directional sense of associated edges MCS curve should be interpreted as opposite the direction its parameterization implies.

In an uncompressed/decoded form, the isUVInc Flag has the following meaning:

= 0	PCS Curve is iso-parameteric in surface parameter domain in one direction and the parameter increases in the other direction
= 1	PCS Curve is iso-parameteric in surface parameter domain in one direction and the parameter decreases in the other direction

The isUVInc flag is set only if the Domain Type of this CoEdge has value between 1 and 4 inclusive.

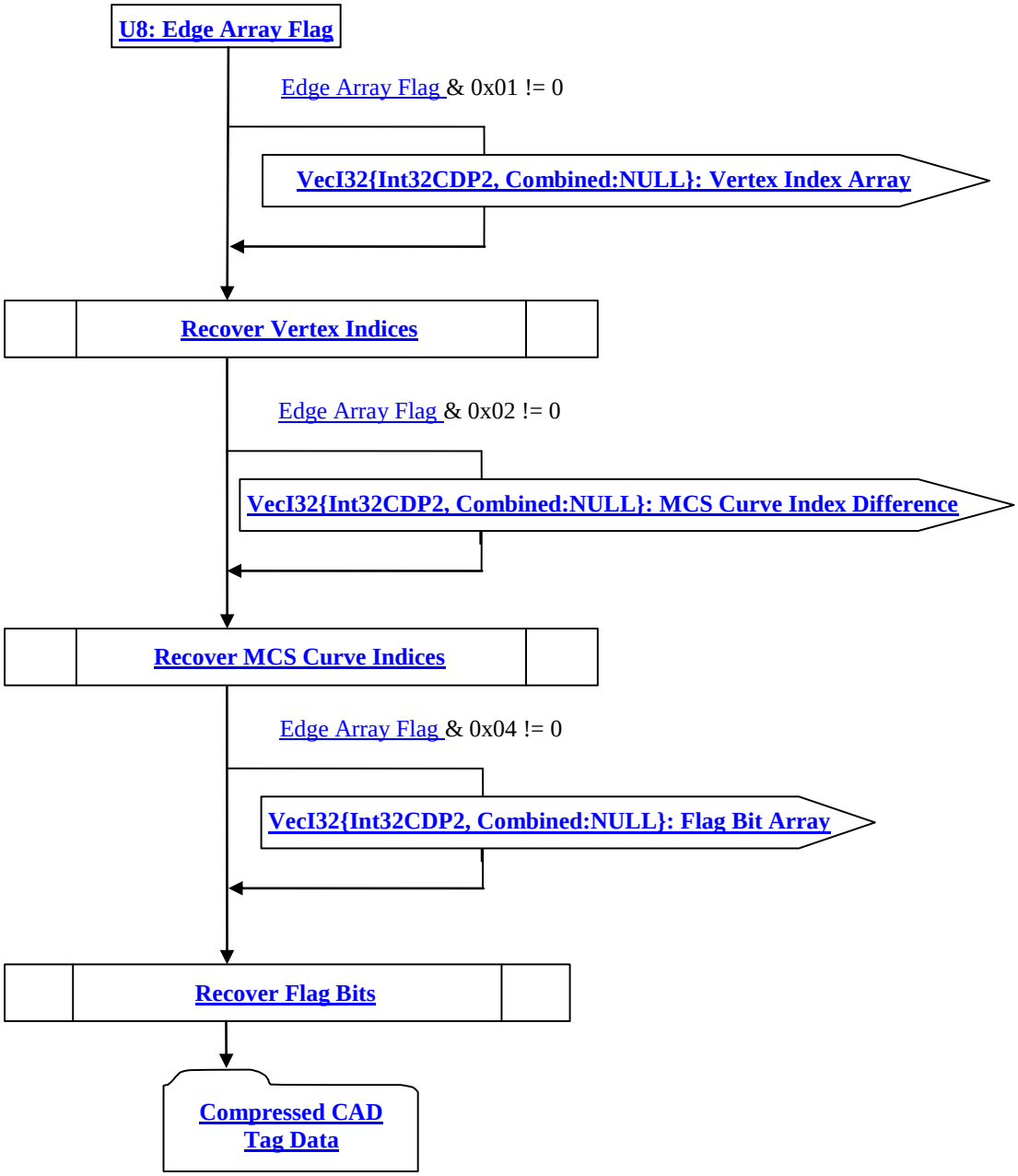
Recover Flag Bits

If CoEdge Array Flag & 0x04 is equal to 0, then each element in Flag Bit Array is set to have value 0. The flag bits are recovered by assigning bits 0~7 of Flag Bit Array to bits 24~31 of Edge Index, and bits 8~15 of Flag Bit Array to bits 24~31 of PCS Curve Index.

7.2.8.1.1.8 Edges Topology Data

An Edge defines a model space trim Loop segment. Edges Topology Data specifies the underlying MCS Curve and start and end Vertex making up each Edge along with an identification tag for each Edge. The complete description of Edge can be found in 7.2.3.1.3.6 Edges Topology Data.

Figure 182: Edges Topology Data collection



U8: Edge Array Flag

Edge Array Flag indicates which arrays of edge topology data are not trivial and therefore encoded.

VecI32{Int32CDP2, Combined:NULL}: Vertex Index Array

Vertex Index Array is a vector of indices representing the start and end vertex indices of each Edge, encoded using Combined Predictor Type. Vertex Index Array is compressed and encoded using the Int32 version of second generation CODEC.

Recover Vertex Indices

If Edge Array Flag & 0x01 is equal to 0, then all the vertex indices of each edge are set to be 0. Otherwise, the start vertex index of Edge with index k is set to be equal to element $2 * k$ of Vertex Index Array, while the end vertex index of this Edge is set to be equal to element $2 * k + 1$ of Vertex Index Array.

VecI32{Int32CDP2, Combined:NULL}: MCS Curve Index Difference

MCS Curve Index Difference is a vector of indices representing the integer value by subtracting the MCS Curve index from the Edge index for each Edge, encoded using Combined Predictor Type. MCS Curve Index Difference is compressed and encoded using the Int32 version of second generation CODEC.

Recover MCS Curve Indices

If Edge Array Flag & 0x02 is equal to 0, then the MCS Curve index of each Edge is equal to the Edge index. Otherwise, the MCS Curve index of Edge with index k can be computed by substracting element k of MCS Curve Index Difference array from k, the Edge index.

VecI32{Int32CDP2, Combined:NULL}: Flag Bit Array

Only the lower 24 bits of the three integer indices, namely MCS Curve index, Start Vertex index, and End Vertex index, are used as integer identifiers. The other bits of these integers are either used to encode additional information, or reserved for future usage.

	24	25	26	27	28	29	30	31
MCS Curve Index	Knot Type		MCS Curve Type		Reserved			
Start Vertex Index	Reserved							
End Vertex Index	Reserved							

The Knot Type, defined in Supported Knot Type, is an integer with its value between 0 and 3.

MCS Curve Type

In an uncompressed/decoded form, the supported MCS Curve types are listed below.

0	Nurbs
1	Line
2	Circle
3	Projection: MCS curve geometry is to be computed from surface geometry and/or PCS curve geometry

Recover Flag Bits

If Edge Array Flag & 0x04 is equal to 0, then each element in Flag Bit Array is set to have value 0. The flag bits are recovered by assigning bits 0~7 of Flag Bit Array to bits 24~31 of MCS Curve Index.

7.2.8.1.1.9 Vertices Topology Data

A Vertex is the simplest topological entity and is basically made up of a geometric Point. Vertices Topology Data specifies the underlying geometric Point making up each Vertex. A Vertex is usually shared/referenced by two or more Edges (e.g. if the corners of four rectangular Faces touches at a common point, this point is represented by a Vertex and is shared by four Edges).

Figure 183: Vertices Topology Data collection

U8: Vertex Array Flag

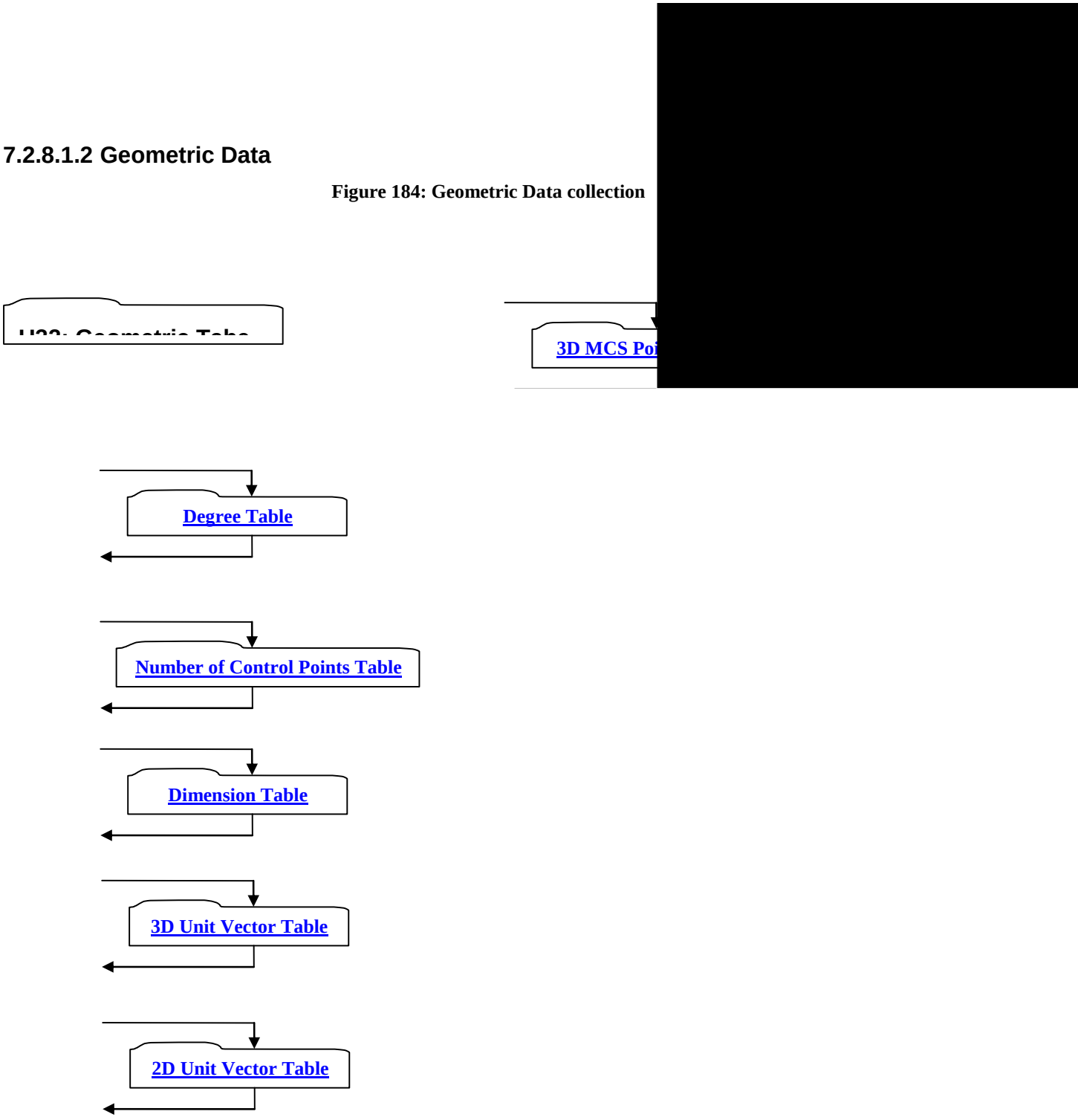
Vertex Array Flag indicates which arrays of vertex topology data are not trivial and therefore encoded.

VecI32{Int32CDP2, Combined:NULL}: Point Index Difference

Point Index Difference is a vector of indices representing the integer value obtained by subtracting point index from vertex index, encoded using Combined Predictor Type. Point Index Difference is

7.2.8.1.2 Geometric Data

Figure 184: Geometric Data collection



CoordF64 : Translation Vector

Translation Vector is a 3-dimensional vector that represents how the ULP geometry is defined w.r.t. the original B-Rep definition from which ULP geometry is derived. If the Translation Vector is not zero vector, then the ULP geometry read from disk is translated from original B-Rep definition by the amount of Translation Vector. This is usually done by the JT writer implementation to improve numerical accuracy of floating point numbers in the ULP geometry. It is important for all the JT readers to take this Translation Vector into consideration when consuming ULP geometry. For example if a LOD is generated from ULP geometry, e.g. by tessellation, then the LOD geometry must be translated to undo the effect of

Translation Vector for it be consistent with the original B-Rep definition. In other words, if we denote the Translation Vector as v, then the LOD geometry from ULP must be translated by $-v$.

U32: Geometric Tabe Flag

Geometric Tabe Flag indicates which geometric tables are not trivial and therefore encoded.

7.2.8.1.2.1 Geometric Entity Counts

U32: Geometric Tabe Flag

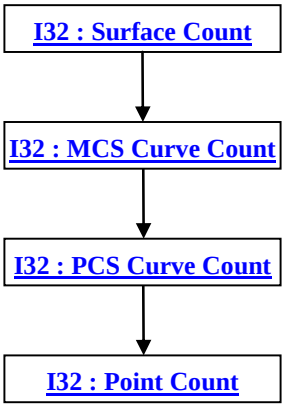
Geometric Tabe Flag indicates which geometric tables are not trivial and therefore encoded.

Geometric Entity Counts data collection defines the counts for each of the various geometric entities within a ULP.

Figure 185: U32: Geometric Tabe Flag

Geometric Tabe Flag indicates which geometric tables are not trivial and therefore encoded.

Geometric Entity Counts data collection



I32 : Surface Count

Surface Count indicates the number of distinct geometric surface entities in the ULP

I32 : MCS Curve Count

MCS Curve Count indicates the number of distinct geometric (Model Coordinate Space) curves (i.e. XYZ curve) entities in the ULP.

I32 : PCS Curve Count

PCS Curve Count indicates the number of distinct geometric Parameter Coordinate Space curves (i.e. UV curve) entities in the ULP

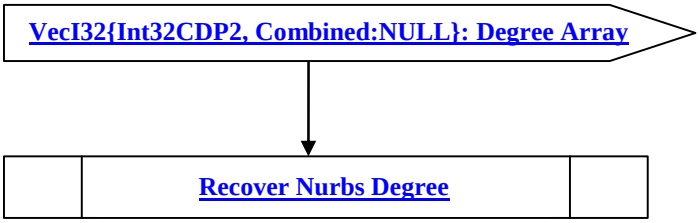
I32 : Point Count

Point Count indicates the number of distinct geometric point entities in the ULP.

7.2.8.1.2.2 Degree Table

Degree Table stores a vector of integers that represent the degree information of Nurbs surfaces and/or curves. If the ULP does not contain any Nurbs entity, then the table is empty and bit 0x0001 in Geometric Tabe Flag is set to be 0.

Figure 186: Degree Table data collection



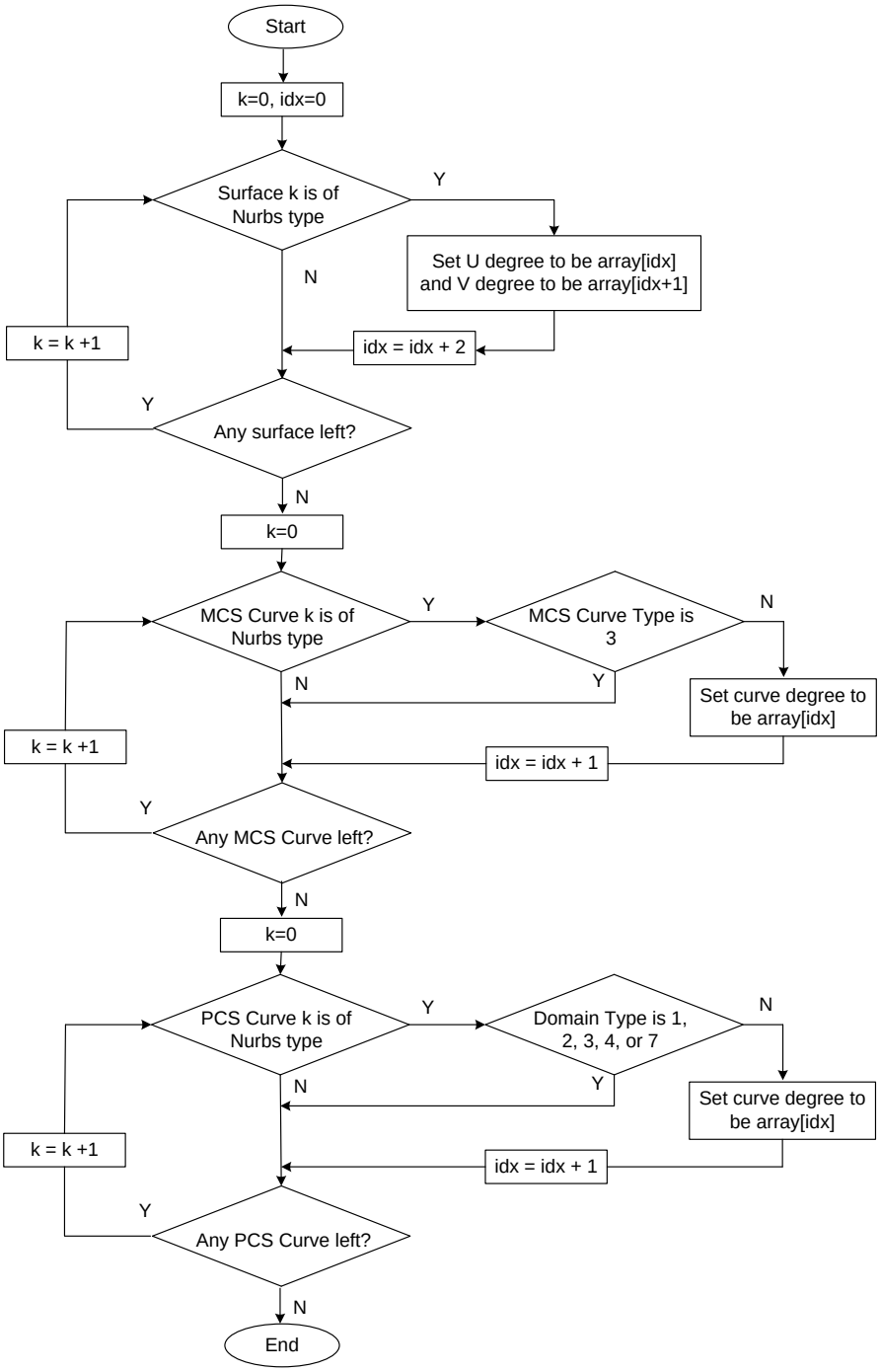
VecI32{Int32CDP2, Combined:NULL}: Degree Array

Degree Array is a vector of integers that stores the degree information for all the Nurbs entities in the ULP, encoded using Combined Predictor Type. Degree Array is compressed and encoded using the Int32 version of second generation CODEC.

Recover Nurbs Degree

The logic diagram to recover degree information for all the Nurbs entities in the ULP from the Degree Array is shown below.

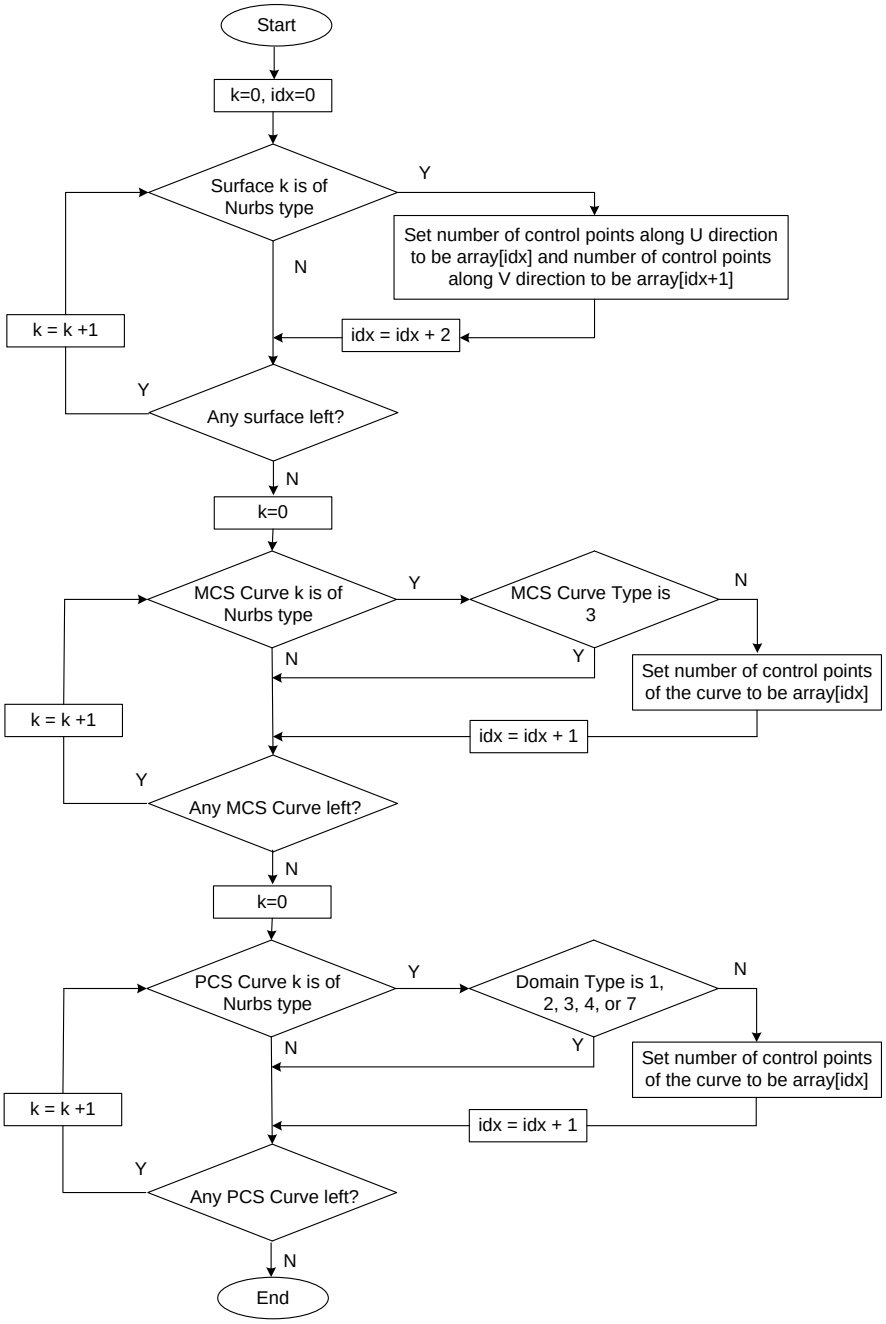
Figure 187: Recover Nurbs Degree



7.2.8.1.2.3 Number of Control Points Table

Number of Control Points Table stores a vector of integers that represent the number of control points information of Nurbs surfaces and/or curves. If the ULP do

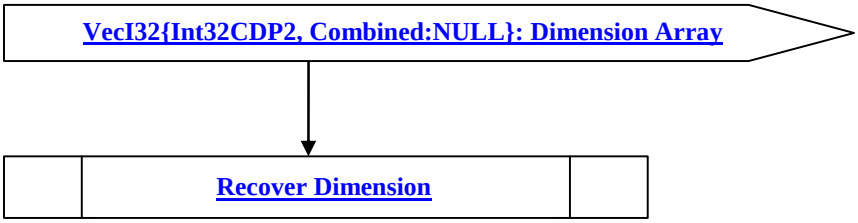
Figure 189: Recover Number of Control Points



7.2.8.1.2.4 Dimension Table

Dimension Table stores a vector of integers that represent the dimension information of Nurbs surfaces and/or curves. If the ULP does not contain any Nurbs entity, then the table is empty and bit 0x0004 in Geometric Tabe Flag is set to be 0.

Figure 190: Dimension Table data collection



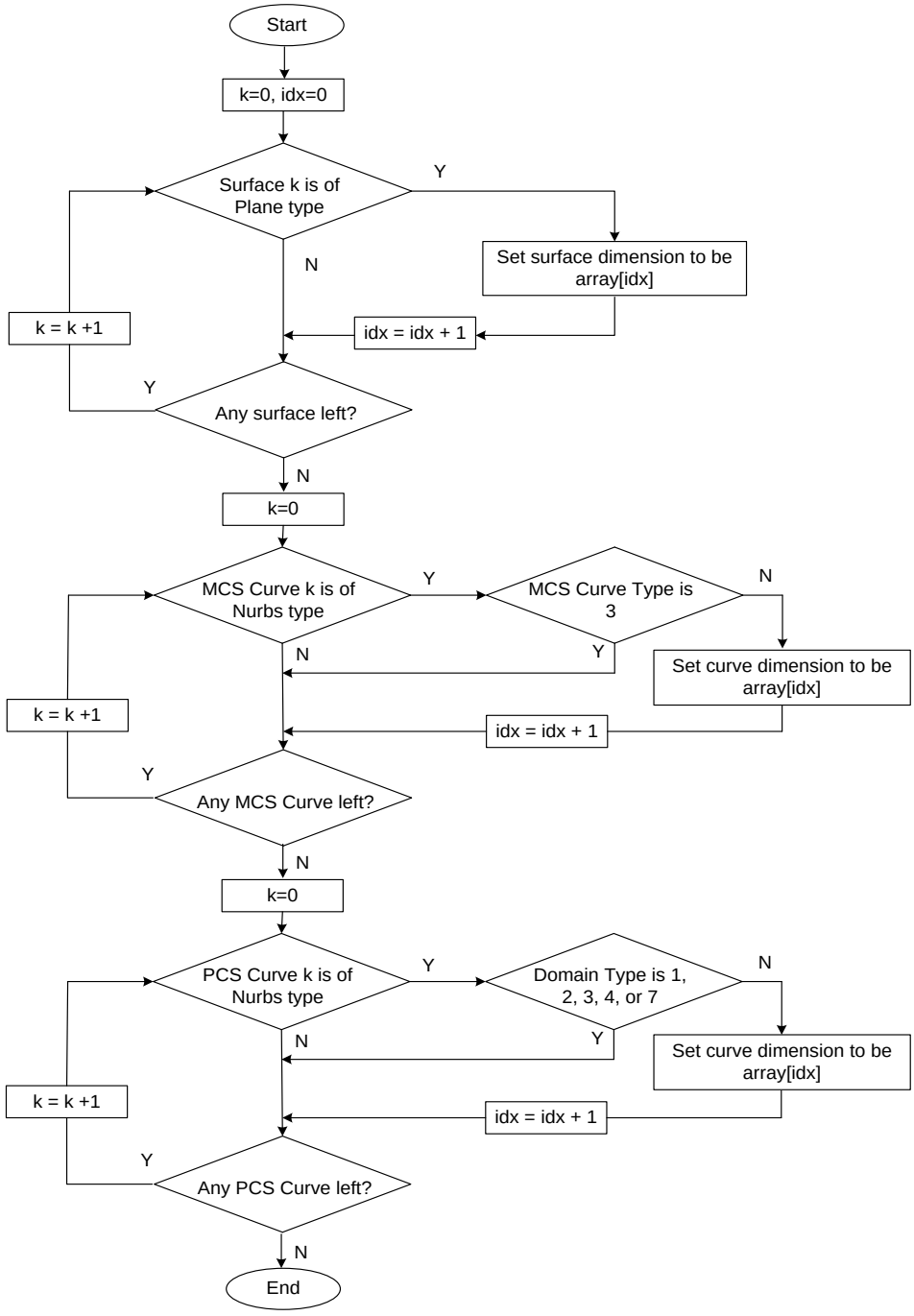
VecI32{Int32CDP2, Combined:NULL}: Dimension Array

Dimension Array is a vector of integers that stores the dimension information for all the Nurbs entities in the ULP, encoded using Combined Predictor Type. Dimension Array is compressed and encoded using the Int32 version of second generation CODEC.

Recover Dimension

The logic diagram to recover dimension information for all the Nurbs entities in the ULP from the Dimension Array is shown below.

Figure 191: Recover Dimension



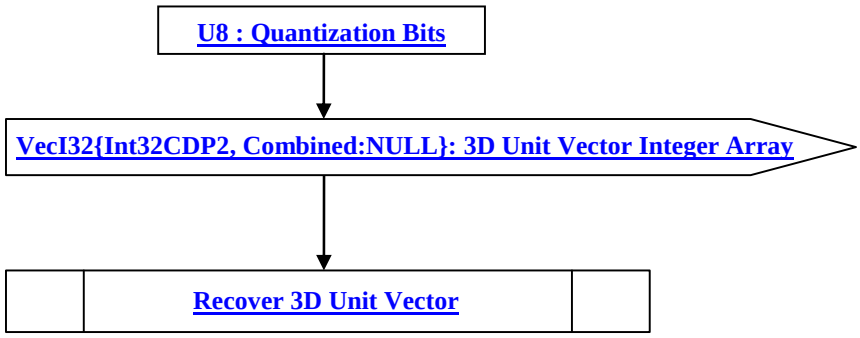
7.2.8.1.2.53D Unit Vector Table

3D Unit Vector Table stores an array of unit vectors in 3D that form part of the analytic surface or curve representation in ULP. If the ULP does not contain any analytic entity, then the table is empty and bit 0x0008 in Geometric Tabe Flag is set to be 0. The supported analytic surface types include plane, cylinder, cone, sphere, and torus, and the supported analytic

curve types include line and circle for both parameter space and model space curves. The analytic representation of ULP follows Parasolid convention as detailed in Appendix F: Parasolid XT Format Reference.

Similar to the coding of 8.1.5 Compressed Vertex Normal Array, each 3D unit vector is encoded as a single 32 bit integer using 8.2.4 Deering Normal CODEC.

Figure 192: 3D Unit Vector Table data collection



U8 : Quantization Bits

The number of bits used for the Deering Normal CODEC if quantization is enabled. A value of 0 denotes that quantization is disabled.

VecI32{Int32CDP2, Combined:NULL}: 3D Unit Vector Integer Array

3D Unit Vector Integer Array is a vector of integers that stores the encoded 3D unit vector from all analytic entities in the ULP, encoded using Combined Predictor Type. 3D Unit Vector Integer Array is compressed and encoded using the Int32 version of second generation CODEC.

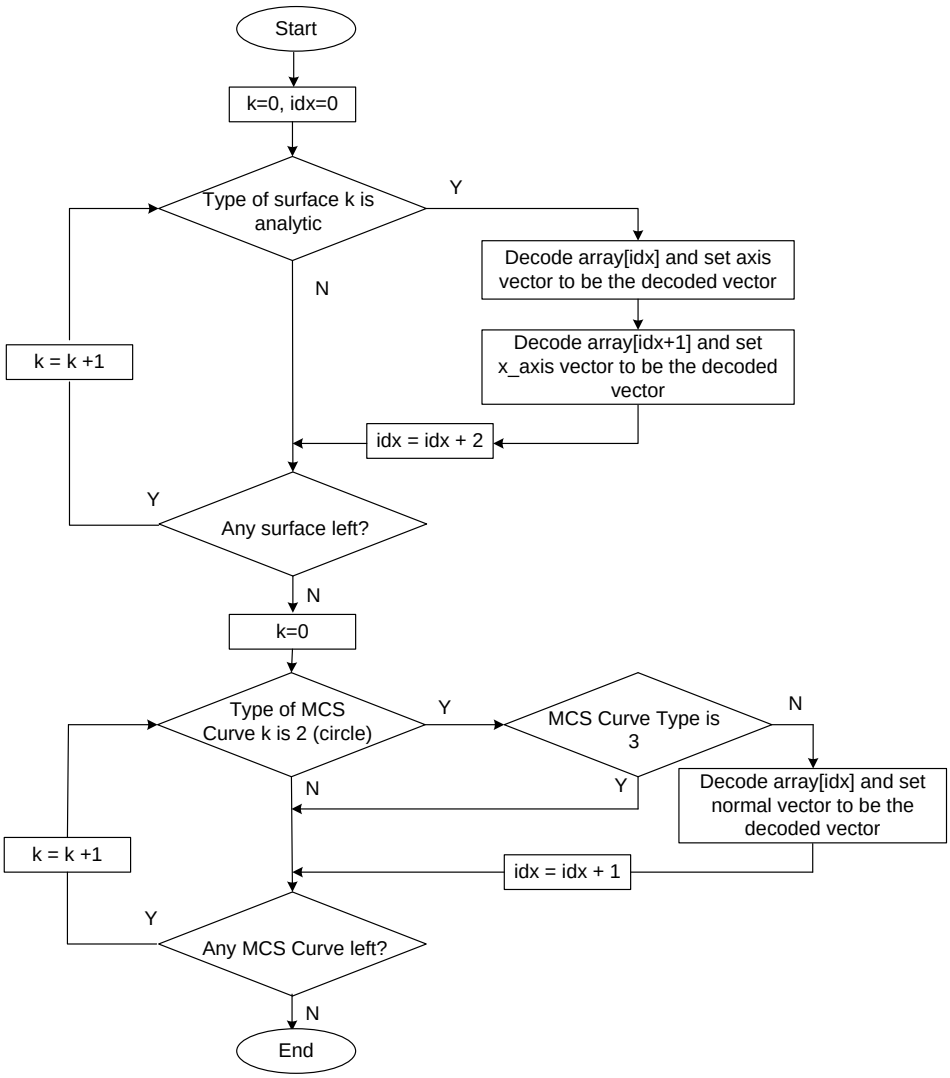
Recover 3D Unit Vector

The logic diagram to recover 3D unit vector information for all the analytic entities in the ULP from the 3D Unit Vector Integer Array is shown below.

The recovery of a unit vector from an element in the 3D Unit Vector Integer Array is done as part of Deering Normal CODEC.

As described in Appendix F: Parasolid XT Format Reference, the representation of an analytic surface of types plane, cylinder, cone, sphere, or torus, includes two 3D unit vectors. 2QH LV FDOOHG ³D[LV´ DQG WKH RWKHU LV FDOOHG ³[BD[LV´ 7KHVH two uQLW YHFWRUV RI HDFK DQD\WLF VXUIDFH DUH UHFRYHUHG IRU HDFK DQD\WLF VXUIDFH ,Q DGGLWLRQ WKH ³QRUPDO´ YHFWRU WR WKH SODQH containing a 3D circle is also recovered.

Figure 193: Recover Dimension

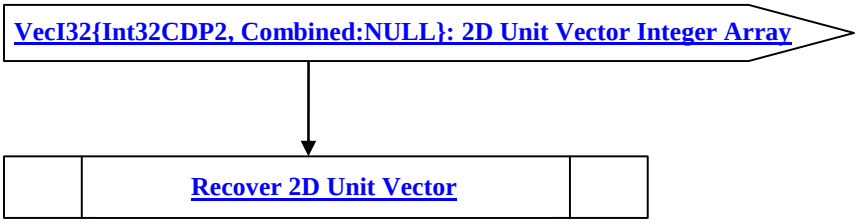


7.2.8.1.2.62D Unit Vector Table

2D Unit Vector Table stores an array of unit vectors in 2D that form part of PCS analytic circle representation in ULP. If the ULP does not contain any analytic circle in the PCS, then the table is empty and bit 0x0010 in Geometric Tabe Flag is set to be 0. The analytic curve representation of ULP follows Parasolid convention as detailed in Appendix F: Parasolid XT Format Reference.

Similar to the coding of 8.1.5 Compressed Vertex Normal Array, each 2D unit vector is treated as a 3D unit vector with z component set to be 0.0, and encoded as a single 32 bit integer using 8.2.4 Deering Normal CODEC. In addition, the Quantization Bits information of Deering Normal CODEC used to encode 2D Unit Vector Table is always the same as the one used for 3D Unit Vector Table.

Figure 194: 2D Unit Vector Table data collection



VecI32{Int32CDP2, Combined:NULL}: 2D Unit Vector Integer Array

2D Unit Vector Integer Array is a vector of integers that stores the encoded 2D unit vector from all analytic entities in the ULP.

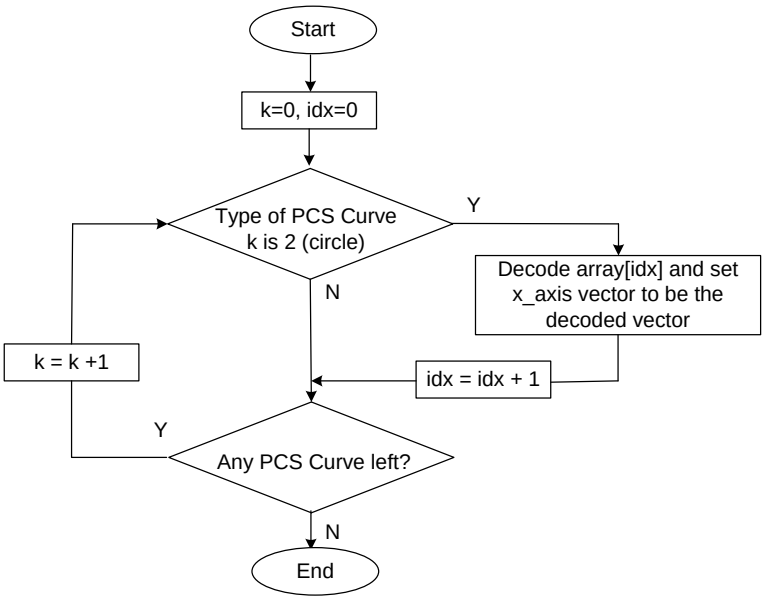
Recover 2D Unit Vector

The logic diagram to recover 2D unit vector information for all the analytic entities in the ULP from the 2D Unit Vector Integer Array is shown below.

The recovery of a unit vector from an element in the 2D Unit Vector Integer Array is done as part of Deering Normal CODEC. The Quantization Bits read from 3D Unit Vector Table should be used for Deering Normal CODEC to decode the vector information from each element in 2D Unit Vector Integer Array.

7KH ³[BD[LV´ YHFWRU WR WKH FLUF0H LQ WKH 3&6 DV described in Appendix F: Parasolid XT Format Reference, is recovered.

Figure 195: Recover 2D Unit Vector

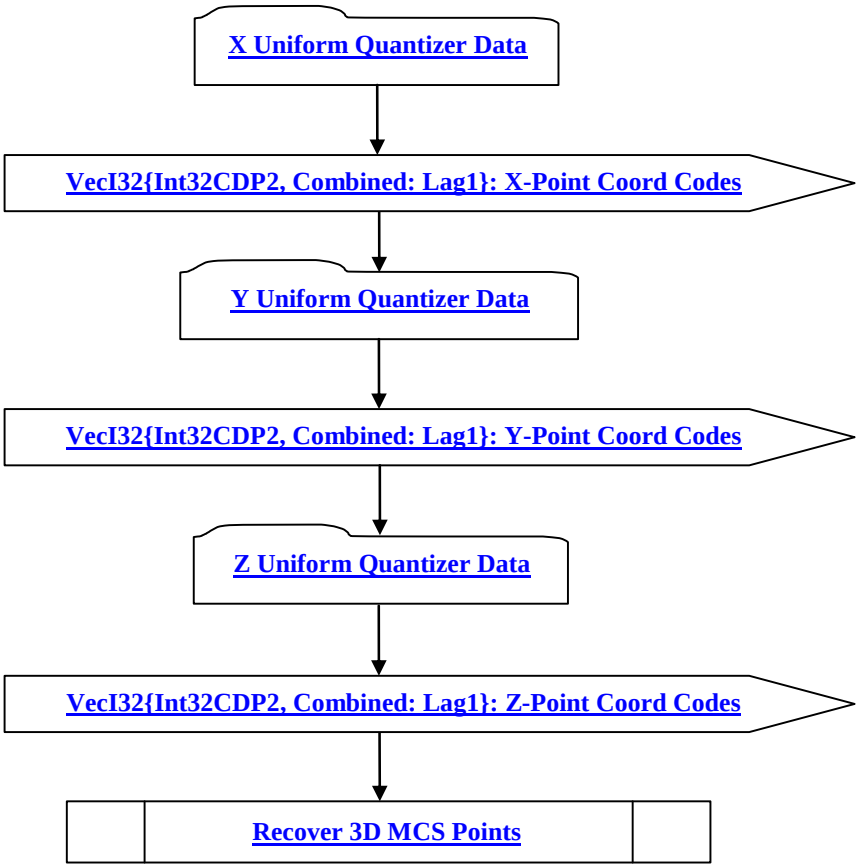


7.2.8.1.2.73D MCS Point Table

3D MCS Point Table stores the quantization representation of an array of 3D MCS points in ULP. If the ULP does not contain 3D MCS points, then the table is empty and bit 0x0020 in Geometric Tabe Flag is set to be 0.

Each point coordinate is first encoded into an integer with uniform quantizer (see 8.1.12 Uniform Quantizer Data) and then all the integers from each coordinate are grouped into an integer array, which is then encoded using the Int32 version of second generation CODEC with Combined Predictor Type.

Figure 196: 3D MCS Point Table data collection



VecI32{Int32CDP2, Combined: Lag1}: X-Point Coord Codes

X-Point Coord Codes LV D YHFWRU RI TXDQWL]HU ³FRGHV´ IRU D00 WKH ;-components of an array of point coordinates. X-Point Coord Codes uses the Int32 version of the second generation CODEC to compress and encode data.

VecI32{Int32CDP2, Combined: Lag1}: Y-Point Coord Codes

Y-Point Coord Codes LV D YHFWRU RI TXDQWL]HU ³FRGHV´ IRU D00 WKH <-components of an array of point coordinates. Y-Point Coord Codes uses the Int32 version of the second generation CODEC to compress and encode data.

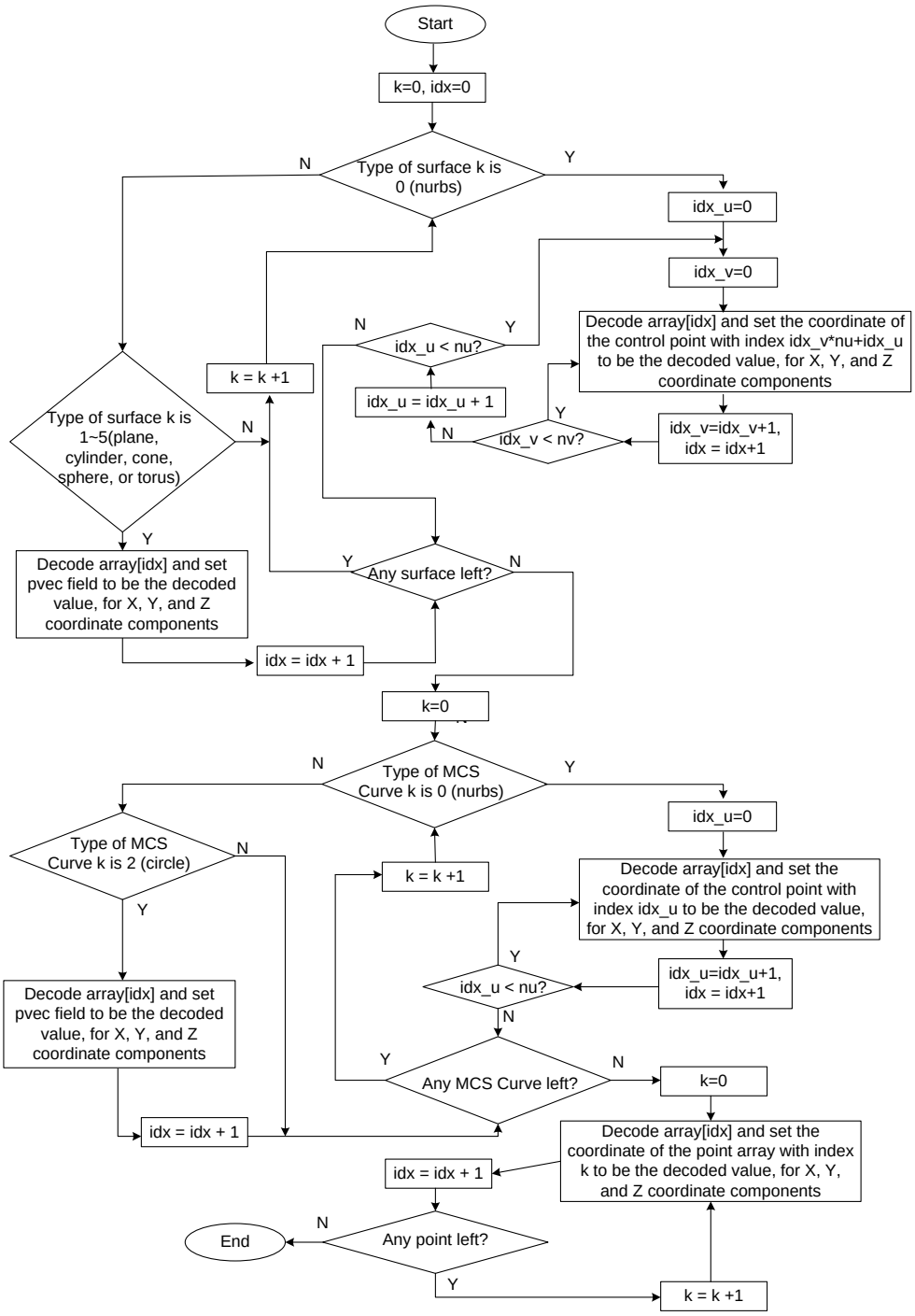
VecI32{Int32CDP2, Combined: Lag1}: Z-Point Coord Codes

Z-Point Coord Codes LV D YHFWRU RI TXDQWL]HU ³FRGHV´ IRU D00 WKH =-components of an array of point coordinates. Z-Point Coord Codes uses the Int32 version of the second generation CODEC to compress and encode data.

Recover 3D MCS Points

The logic diagram to recover 3D MCS points information in the ULP from the three arrays, X-Point Coord Codes, Y-Point Coord Codes, and Z-Point Coord Codes, is shown below. Note that the point coordinates are decoded from the integer elements with Uniform Quantizer (see 8.1.12 Uniform Quantizer Data).

Figure 197: Recover 3D MCS Points



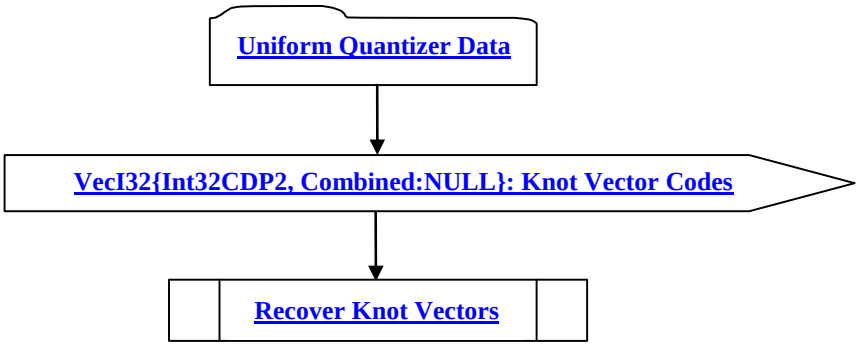
7.2.8.1.2.8 Knot Vector Table

Knot Vector Table stores the quantization representation of knot vectors in ULP. If the ULP does not contain any knot vector that needs be stored, then the table is empty and bit 0x0040 in Geometric Tabe Flag is set to be 0.

In ULP every knot vector starts with 0.0 and ends with 1.0 and is always clamped at both ends. The encoding of knot vector depends on its classified knot type. The knot values in the middle of a knot vector need be written only if the knot type is 0

(see Supported Knot Type). For all the knot values that need be written, each of them is encoded into an integer with uniform quantizer (see 8.1.12 Uniform Quantizer Data) and then all the integers are grouped into an integer array. The integer array is then encoded using the Int32 version of second generation CODEC with Combined Predictor Type.

Figure 198: Knot Vector Table data collection



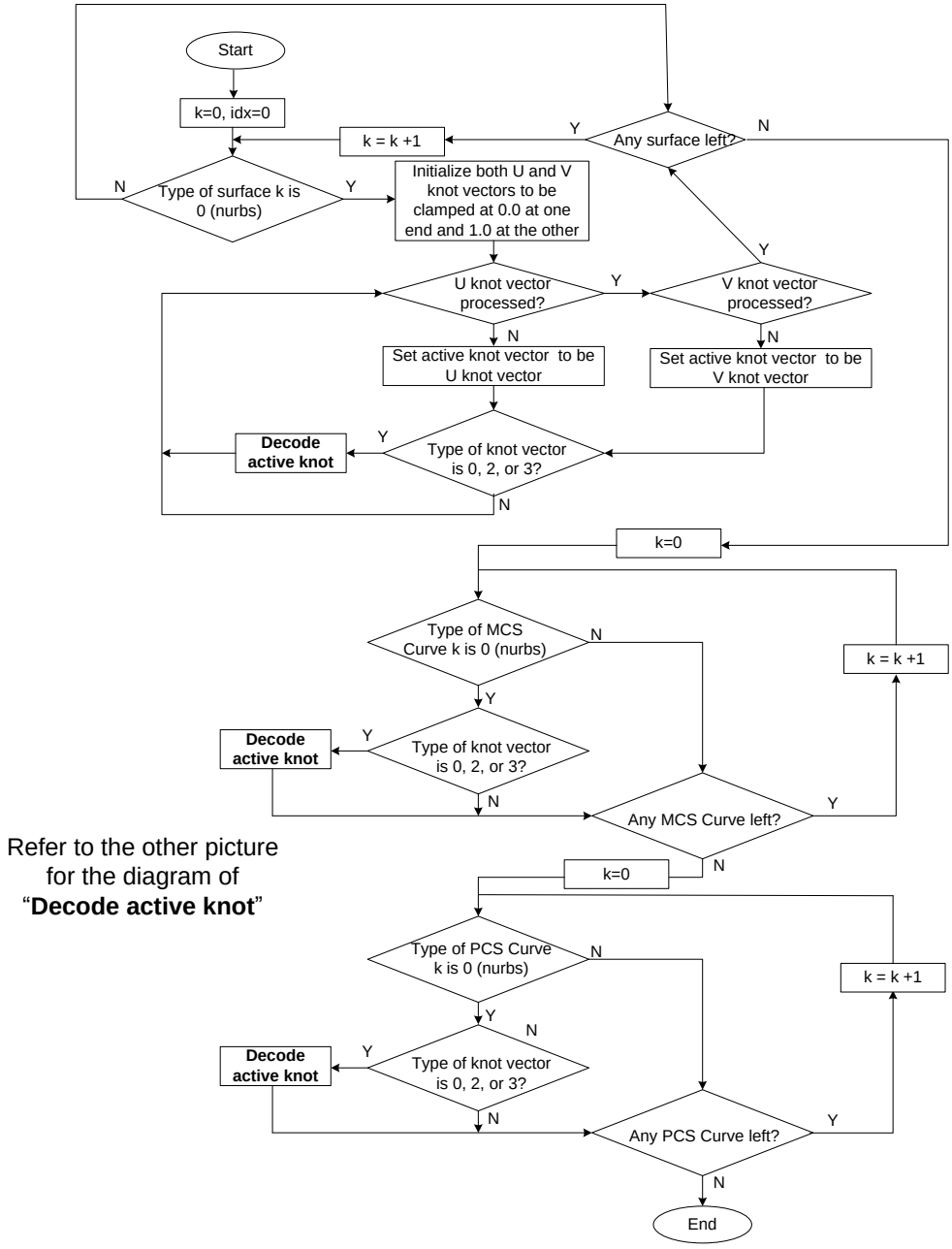
VecI32{Int32CDP2, Combined:NULL}: Knot Vector Codes

Knot Vector Codes LV D YHFWRU RI TXDQWL]HU ³FRGHV´ IRU DOO WKH NQRW YHFWRrs. Knot Vector Codes uses the Int32 version of the second generation CODEC with Combined Predictor Type to compress and encode data.

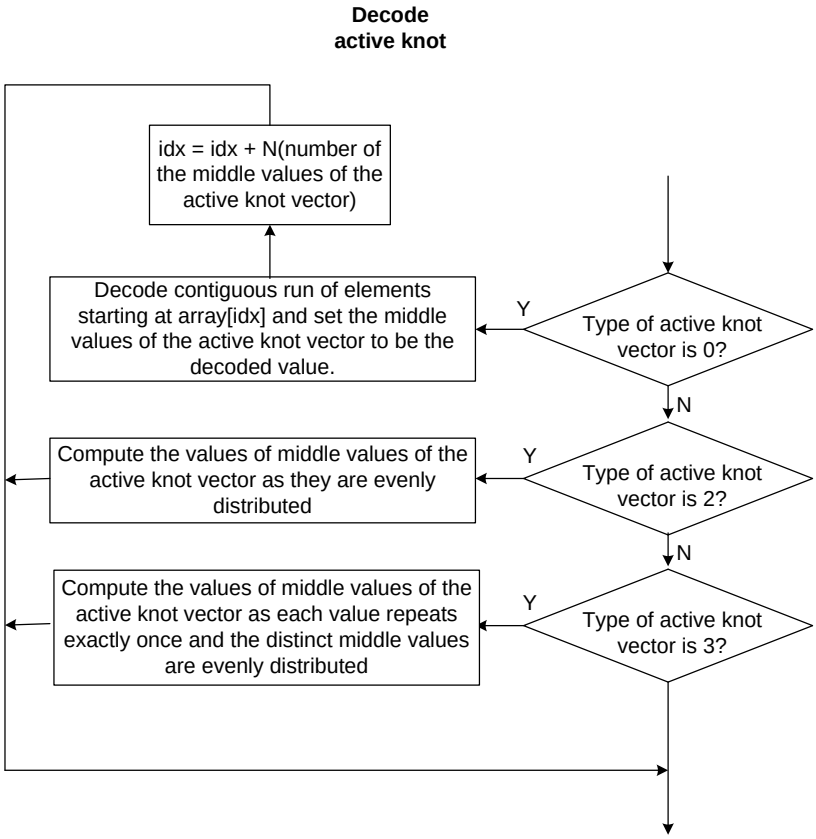
Recover Knot Vectors

The logic diagram to recover knot vector information in the ULP from the Knot Vector Codes is shown below. Note that each integer element in the Knot Vector Codes array is decoded with Uniform Quantizer.

Figure 199: Recover Knot Vectors



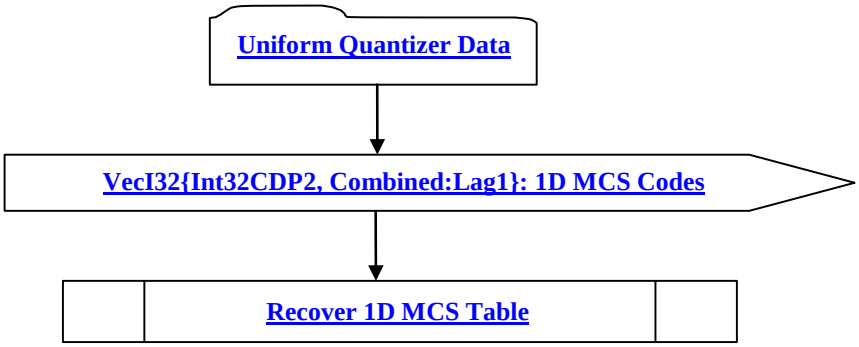
Refer to the other picture for the diagram of "**Decode active knot**"



7.2.8.1.2.91D MCS Table

1D MCS Table stores the quantization representation of floating point values in MCS. If the ULP does not contain any such value, then the table is empty and bit 0x0080 in Geometric Tabe Flag is set to be 0. Each floating point value is encoded into an integer with uniform quantizer (see 8.1.12 Uniform Quantizer Data) and then all the integers are grouped into an integer array. The integer array is then encoded using the Int32 version of second generation CODEC with Combined Predictor Type.

Figure 200: 1D MCS Table data collection



Vecl32{Int32CDP2, Combined:Lag1}: 1D MCS Codes

1D MCS Codes LV D YHFWRU RI TXDQWL]HU ³FRGHV´ IRU DOO WKH ' IORDWLQg point values in MCS . 1D MCS Codes uses the Int32 version of the second generation CODEC with Combined Predictor Type to compress and encode data.

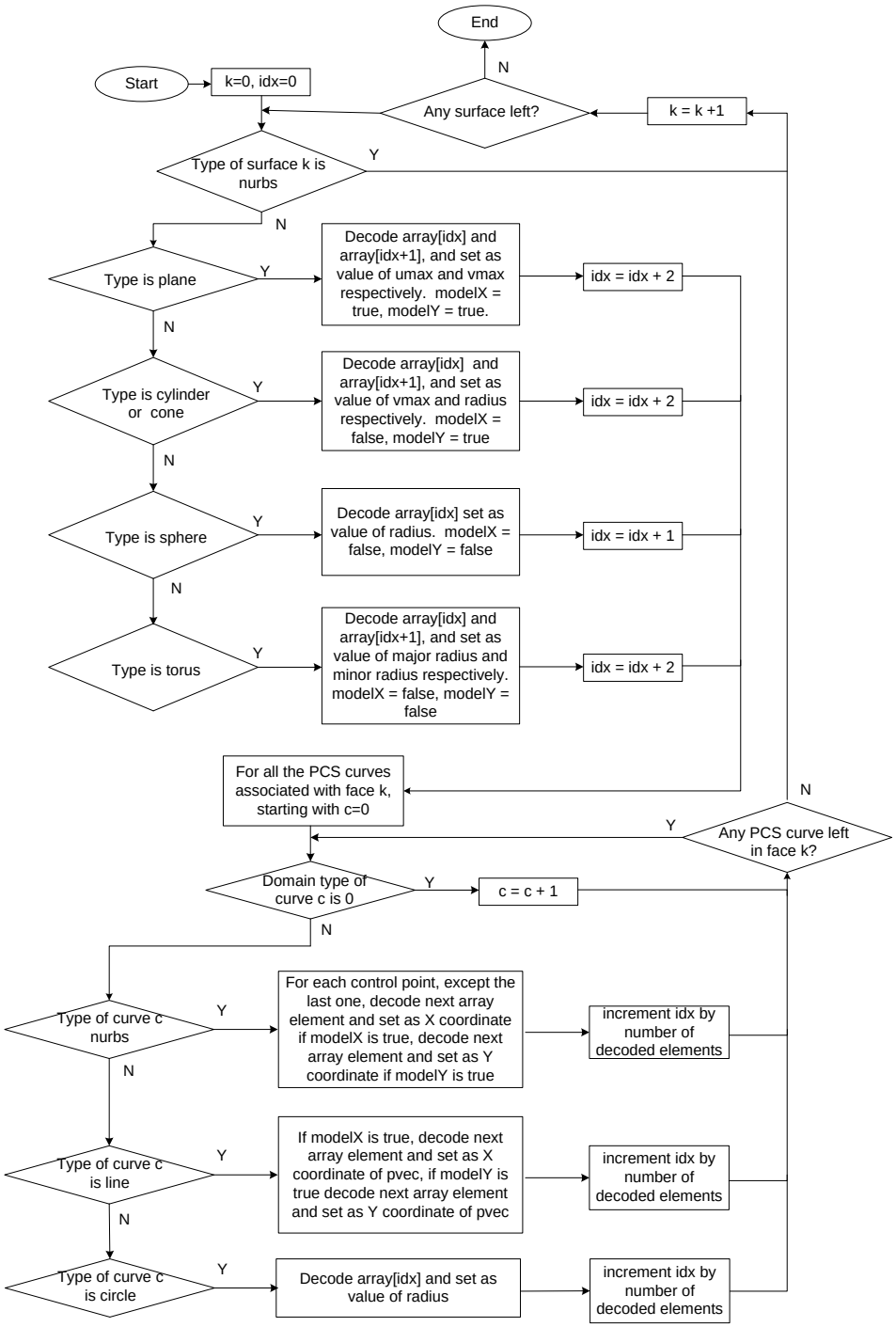
Recover 1D MCS Table

The representation of each surface or curve in ULP includes information that describes the extent of the surface or curve in the parameter domain. For curves the extent information is represented by two numbers, umin and umax, while for surfaces it is represented by two additional numbers for the other parametric direction, vmin and vmax. For surfaces or curves of NURBS type such extent information is implied by the knot vector information. For surfaces or curves of other types the extent information needs be read from 1D MCS Table if the parameter value represents value in MCS, or Radian Table if the parameter value represents angle information. The detailed information about how the parameter domain information of different entities should be read is listed in Table 8.

Table 8: Parameter Domain

Entity Type	umin	umax	vmin	vmax
NURBS Surface	n/a (from knot)	n/a (from knot))	n/a (from knot)	n/a (from knot)
Plane	n/a (always 0)	1D MCS Table	n/a (always 0)	1D MCS Table
Cylinder	n/a (always 0)	Radian Table	n/a (always 0)	1D MCS Table
Cone	n/a (always 0)	Radian Table	n/a (always 0)	1D MCS Table
Sphere	n/a (always 0)	Radian Table	Radian Table	Radian Table
Torus	n/a (always 0)	Radian Table	Radian Table	Radian Table
XYZ NURBS Curve	n/a (from knot)	n/a (from knot)	n/a	n/a
XYZ Line	n/a (always 0)	n/a (from vertex geometry)	n/a	n/a
XYZ Circle	n/a (always 0)	Radian Table	n/a	n/a
UV NURBS Curve	n/a (from knot)	n/a (from knot)	n/a	n/a
UV Line	n/a (always 0)	n/a (from next uv curve)	n/a	n/a
UV Circle	Radian Table	Radian Table	n/a	n/a

Figure 201: Recover 1D MCS Table

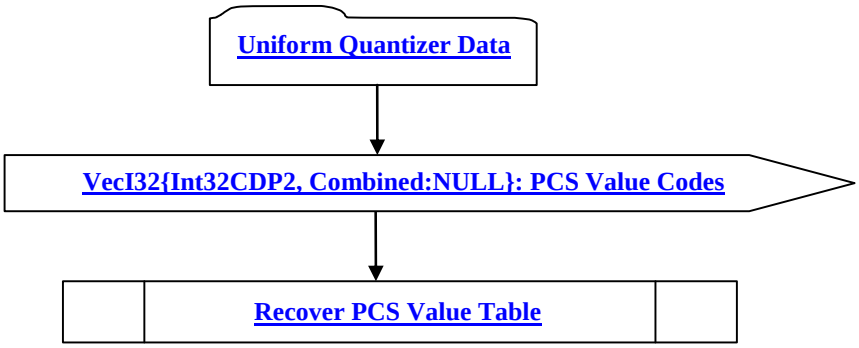


The logic diagram to recover 1D MCS table information in the ULP from the 1D MCS Codes is shown in igure 200: 1D MCS Table data collectionFigure 201. Note that each integer element in the 1D MCS Codes array is decoded with Uniform Quantizer.

7.2.8.1.2.10 PCS Value Table

PCS Value Table stores the quantization representation of floating point values in PCS. If the ULP does not contain any such value, then the table is empty and bit 0x0100 in Geometric Tabe Flag is set to be 0. Each floating point value is encoded into an integer with uniform quantizer (see 8.1.12 Uniform Quantizer Data) and then all the integers are grouped into an integer array. The integer array is then encoded using the Int32 version of second generation CODEC with Combined Predictor Type.

Figure 202: PCS Value Table data collection



VecI32{Int32CDP2, Combined:NULL}: PCS Value Codes

PCS Value Codes LV D YHFWRU RI TXDQWL]HU ³FRGHV´ IRU DOO WKH IORDWLQJ SRLQW YDOXHV in PCS . PCS Value Codes uses the Int32 version of the second generation CODEC with Combined Predictor Type to compress and encode data.

Recover PCS Value Table

The logic diagram to recover PCS Value Table information in the ULP from the PCS Value Codes is shown in Figure 203. Note that each integer element in the PCS Value Codes array is decoded with Uniform Quantizer.

Figure 203: Recover PCS Value Table

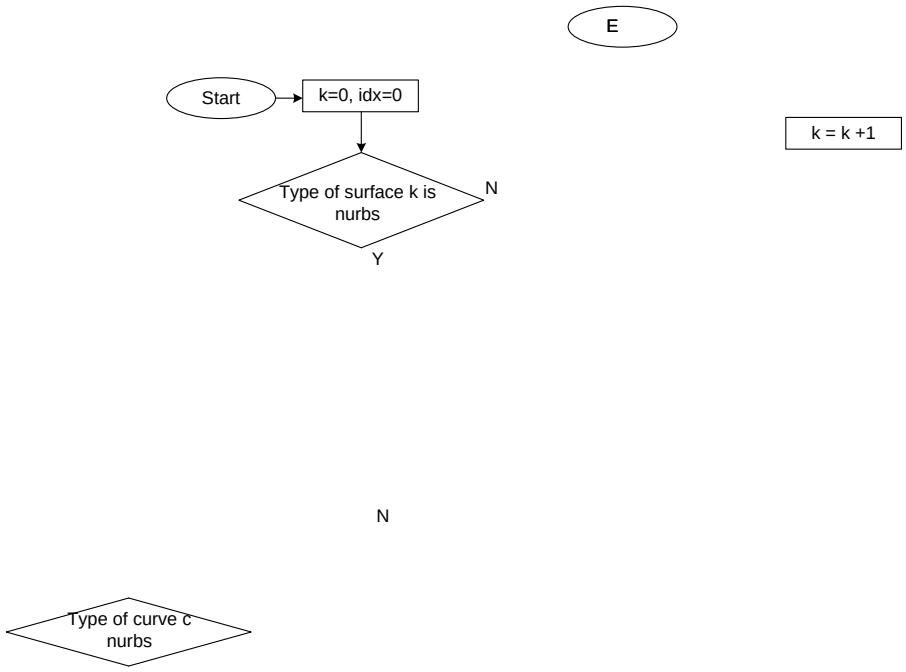
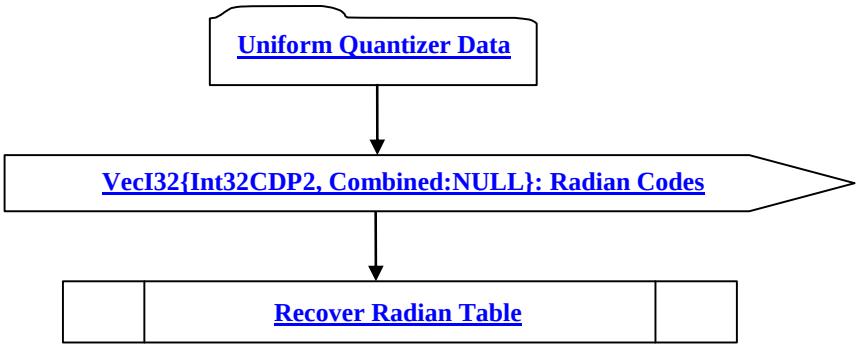


Figure 204: Radian Table data collection



7.2.8.1.2.11 Radian Table

Radian Table stores the quantization representation of angular values. If the ULP does not contain any such angular value, then the table is empty and bit 0x0200 in Geometric Tabe Flag is set to be 0. Each angular value is encoded into an integer with uniform quantizer (see 8.1.12 Uniform Quantizer Data) and then all the integers are grouped into an integer array. The integer array is then encoded using the Int32 version of second generation CODEC with Combined Predictor Type.

Vecl32{Int32CDP2, Combined:NULL}: Radian Codes

Radian Codes LV D YHFWRU RI TXDQWL]HU ³FRGHV´ IRU D00 WKH DQJX0DU YD0XHV Radian Codes uses the Int32 version of the second generation CODEC with Combined Predictor Type to compress and encode data.

Recover Radian Table

The logic diagram to recover Radian Table information in the ULP from the Radian Codes is shown in Figure 205. Note that each integer element in the Radian Codes array is decoded with Uniform Quantizer.

Figure 205: Recover Radian Table

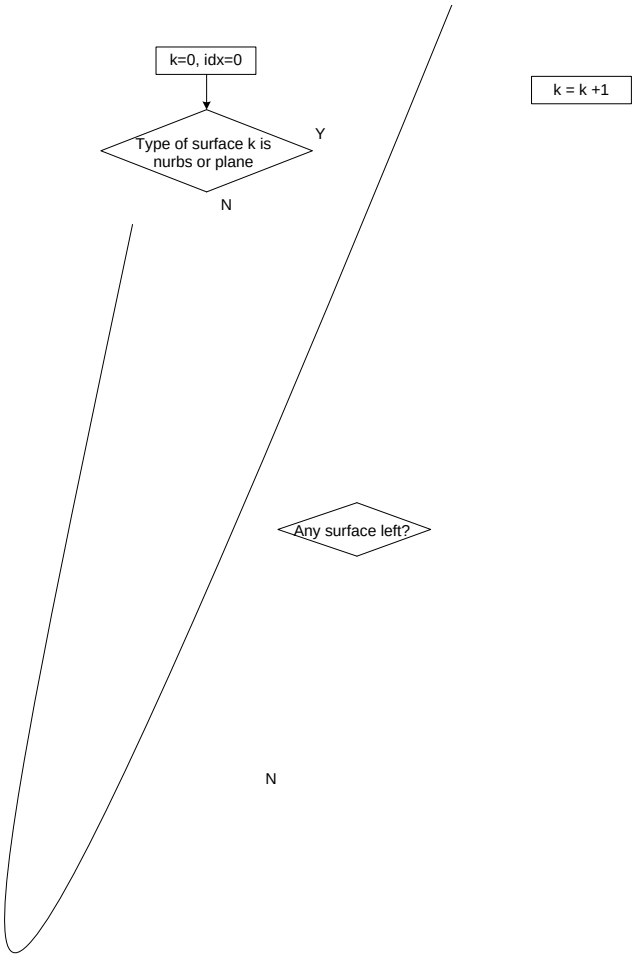
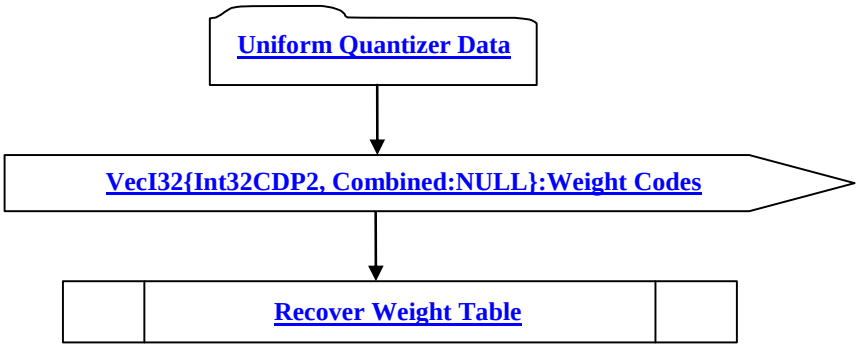


Figure 206: Weight Table data collection



7.2.8.1.2.12 Weight Table

Weight Table stores the quantization representation of weight values. If the ULP does not contain any such weight value, then the table is empty and bit 0x0400 in Geometric Tabe Flag is set to be 0. Each weight value is encoded into an integer with uniform quantizer (see 8.1.12 Uniform Quantizer Data) and then all the integers are grouped into an integer array. The integer array is then encoded using the Int32 version of second generation CODEC with Combined Predictor Type.

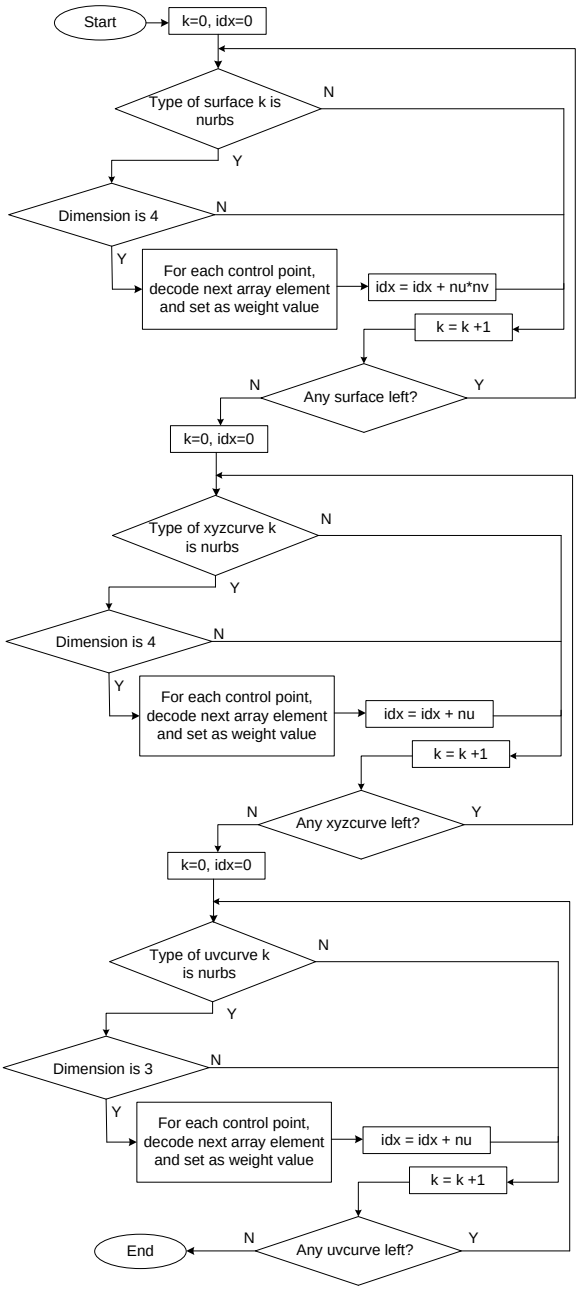
VecI32{Int32CDP2, Combined:NULL}:Weight Codes

Weight Codes LV D YHFWRU RI TXDQWL]HU ³FRGHV´ IRU DOO WKH ZHLJKW YDOXHV Weight Codes uses the Int32 version of the second generation CODEC with Combined Predictor Type to compress and encode data.

Recover Weight Table

The logic diagram to recover Weight Table information in the ULP from the Weight Codes is shown in Figure 207. Note that each integer element in the Weight Codes array is decoded with Uniform Quantizer.

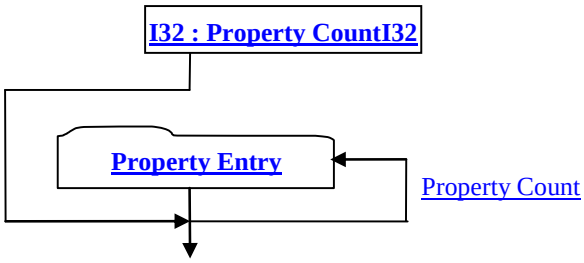
Figure 207: Recover Weight Table



7.2.8.1.3 Material Attribute Element Properties

The properties attached to material attribute are standard JT properties, and the logic diagram to read the properties attached a material attribute is shown in Figure 208.

Figure 208: Material Attribute Element Properties



I32 : Property Count

Property count is the number of properties attached.

Property Entry

Standard JT property entry, consisting of key and value pair.

7.2.8.1.4 Information Recovery

7KH LQIRUPDWLRQ LQ 8/3 LV FODVVLILHG DV ³HVVHQWLDO LQIRUPDWLRQ´ WKDW LV H[SOLFLWO\ ZULWWHQ RQ GLVN DQG ³GHULYDWLYH LQIRUPDWLRQ´ that can be computed from the ³HVVHQWLDO LQIRUPDWLRQ´ +RZ ³HVVHQWLDO LQIRUPDWLRQ´ RI 8/3 FDQ EH UHDG IURP GLVN ZDV covered in previous sections, and this section focuses RQ WKH ORJLF WR UHFRYHU ³GHULYDWLYH LQIRUPDWLRQ´ IURP ³HVVHQWLDO LQIRUPDWLRQ´

The derivative information consists of curve information either in the parameter or model space. For example, the PCS curves associated with an untrimmed face can be inferred from the parameter domain of the surface, or an MCS curve may be computed from vertex information and/or the combination of corresponding PCS curve geometry and surface geometry, etc.. Shown in Figure 209 LV WKH KLJK OHYHO GLDJUDP WR UHFRYHU ³GHULYDWLYH LQIRUPDWLRQ´)LUVW DOl the PCS line geometry are recovered from the associated surface domain information if the domain type of those PCS curves, stored in its associated coedge, are of value 1, 2, 3, 4 meaning that the PCS curve is identical to one of the parameter boundaries of the surface. Second, the MCS curve geometry is recovered depending on its type. If the MCS curve type is 0, 1, or 2, then the geometry of its two end vertices is used to compute the curve geometry. If the MCS curve type is 3, then its geometry is computed by projecting PCS curve onto the surface geometry. After all the MCS curve geometry is computed, all the PCS curves of type 7 is computed by projecting MCS curve onto the parameter domain. Some part of PCS curve definition may still be missing after all these steps. At the end, all the information that is still missing in some of the PCS curves is recovered by leveraging the knowledge that all PCS curves within the same loop are connected in a head to tail fashion. The logical steps that are displayed with dark color indicate steps that will be elaborated in more detail later.

Figure 209: Information Recovery

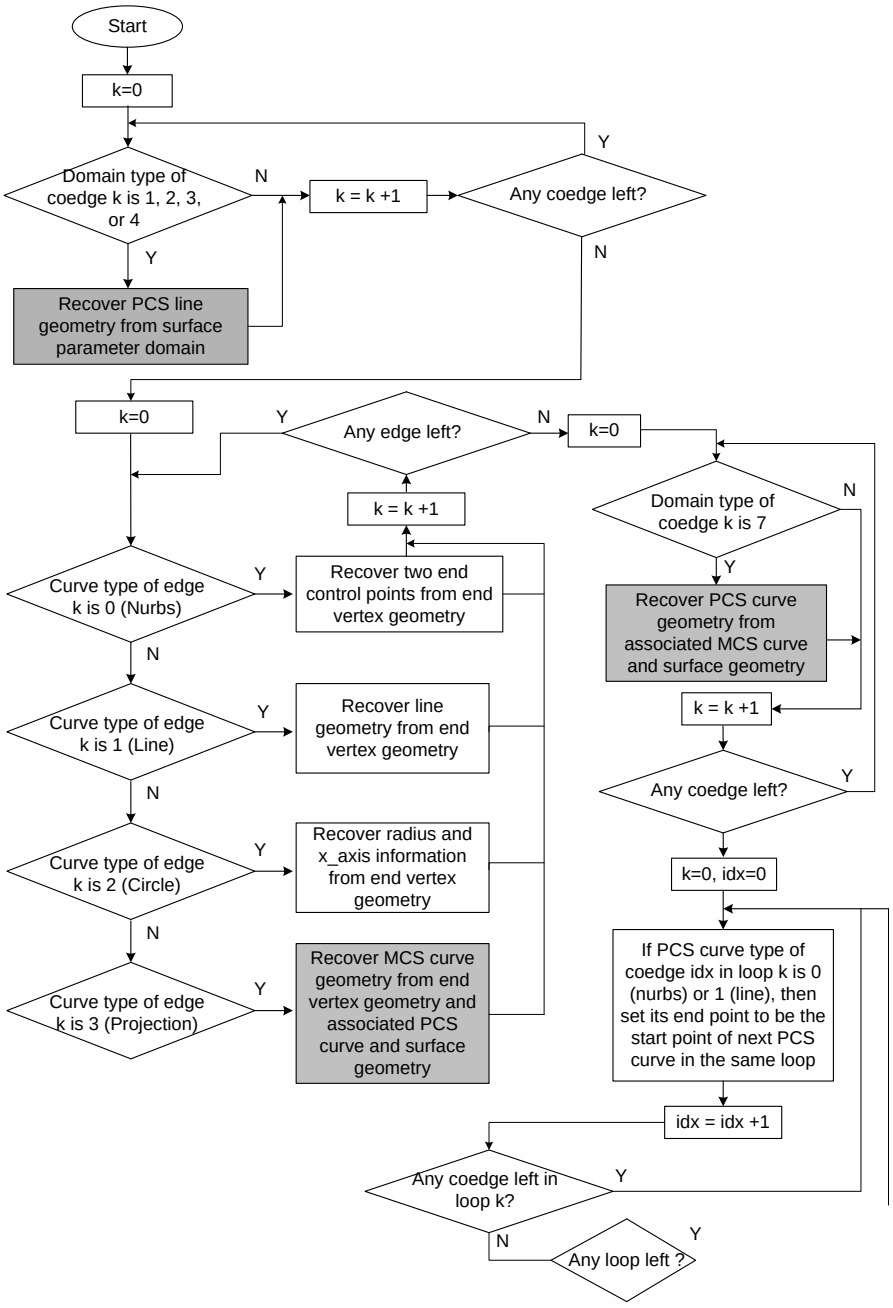
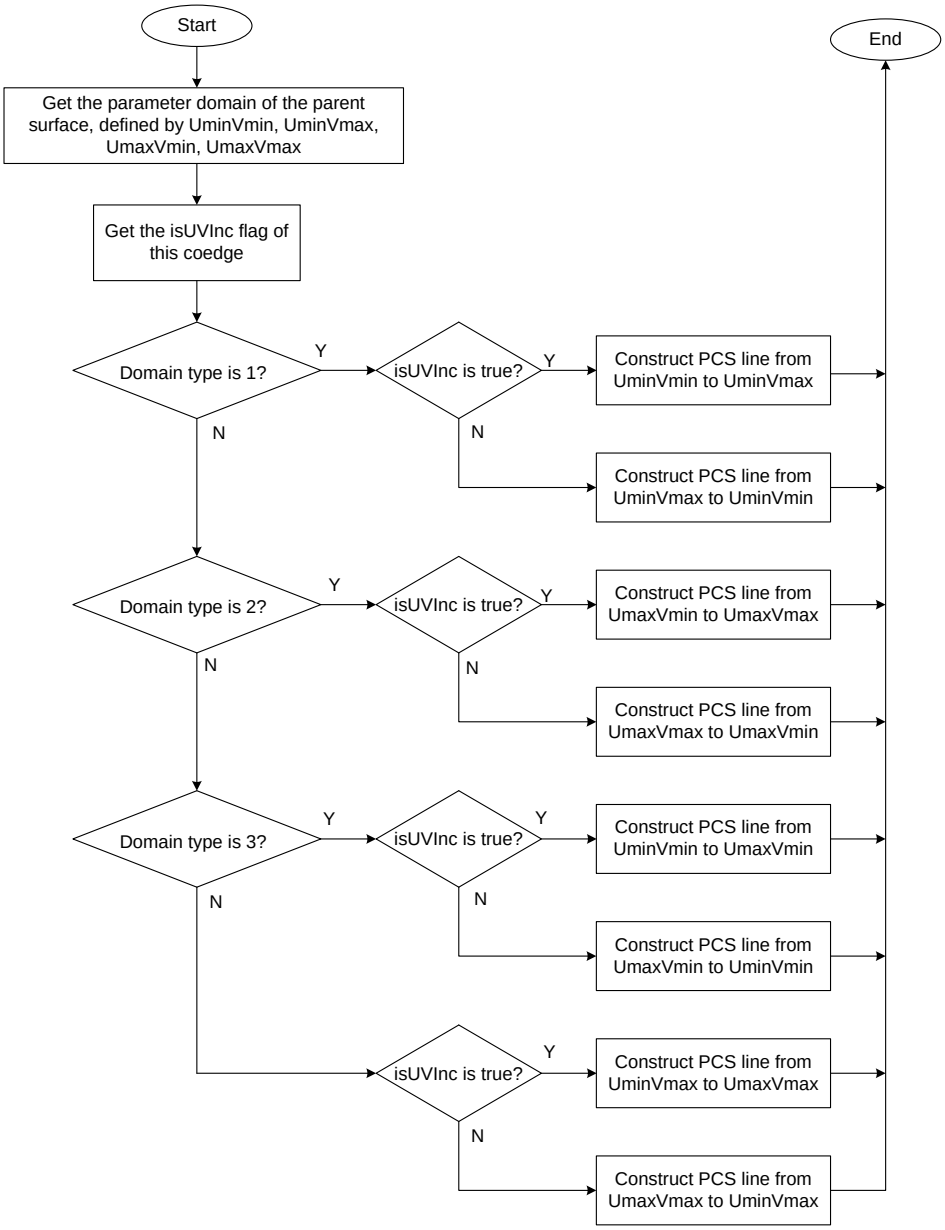


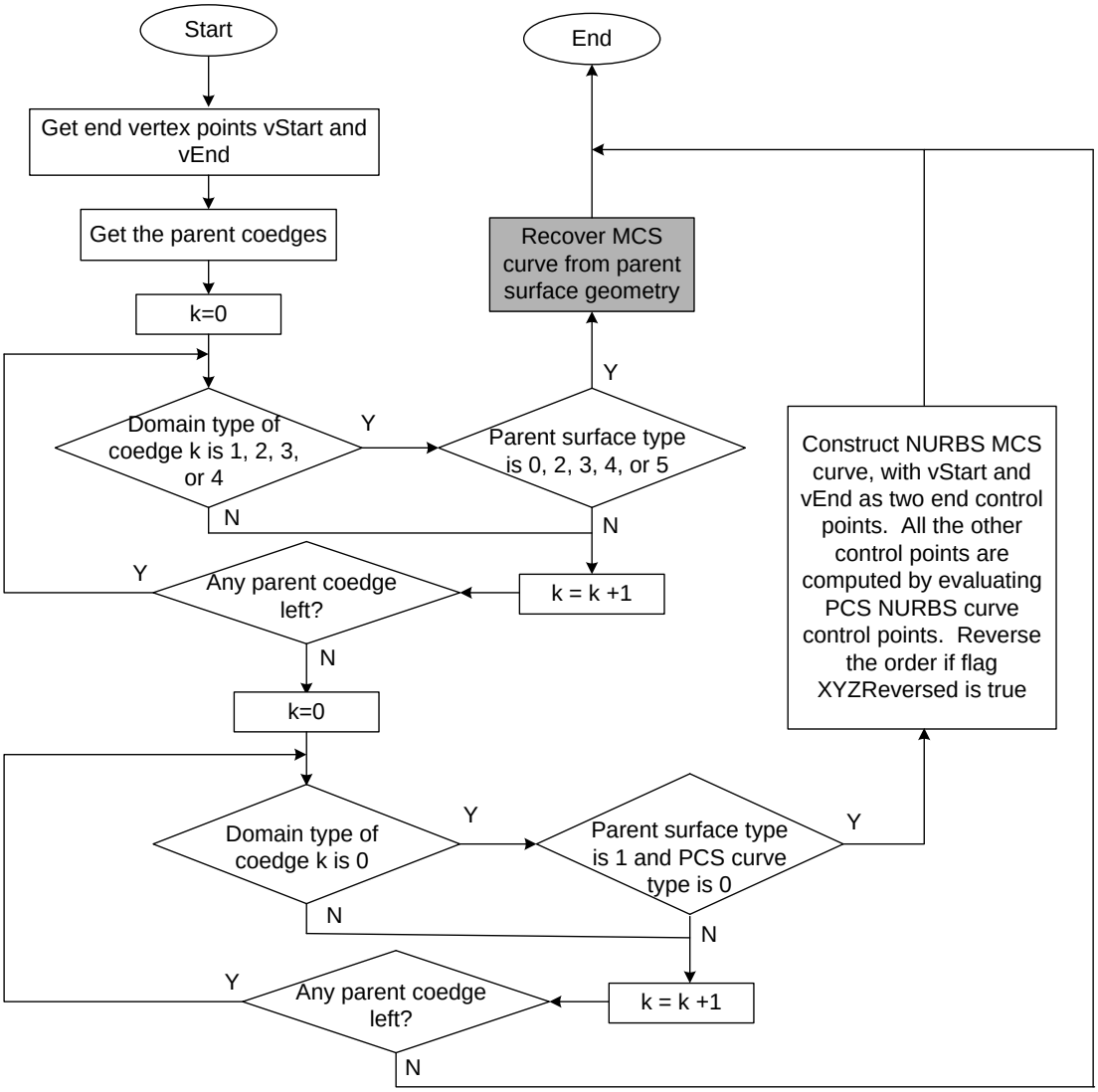
Figure 210: PCS Curve Recovery from Surface Domain



7.2.8.1.4.2 MCS Curve Recovery

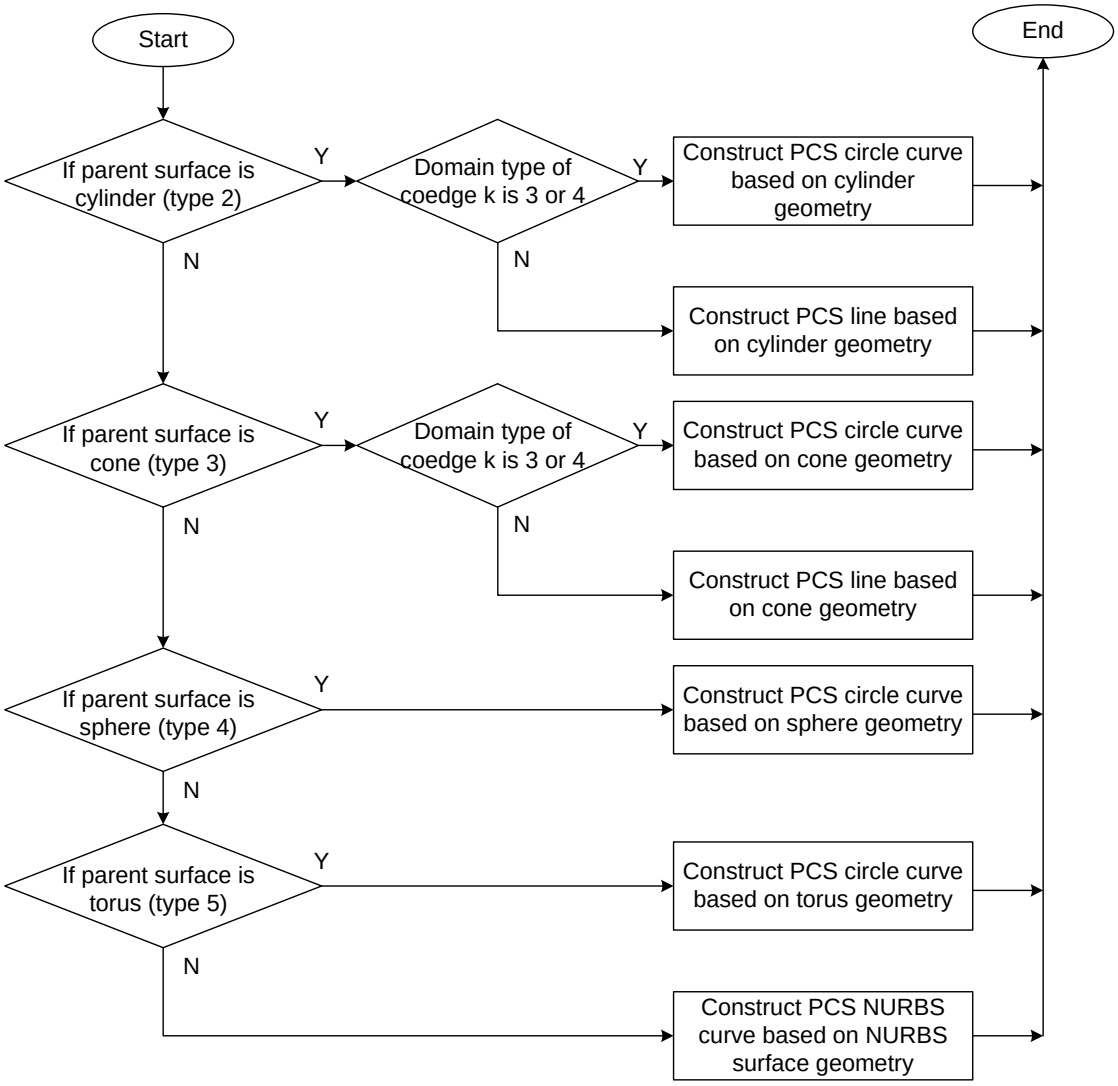
Shown in Figure 211 is the diagram illustrating how MCS curve geometry is recovered from its end vertex geometry, and/or its associated PCS curve geometry and surface geometry. If the associated PCS curve is coincident with one of the parameter boundaries of the parent surface, then the MCS curve can be recovered from parent surface geometry. Otherwise, if the surface type is planar and PCS curve is of type NURBS, then the MCS curve geometry can be recovered by projecting the PCS curve from parameter domain to model space onto the planar surface.

Figure 211: MCS Curve Recovery



Shown in Figure 212 is the detailed description of how MCS curve can be recovered from surface geometry.

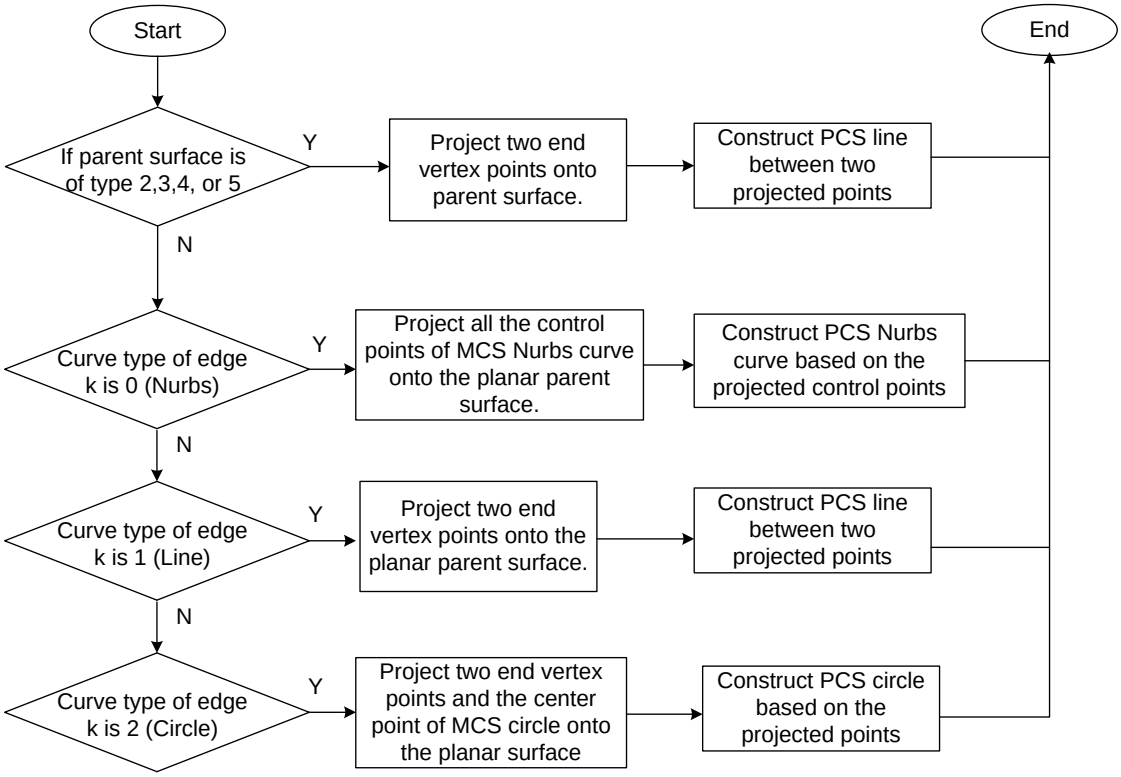
Figure 212: MCS Curve Recovery from Surface Geometry



7.2.8.1.4.3 PCS Curve Recovery from MCS Curve and Surface Geometry

Shown in Figure 213 is the diagram illustrating how PCS curve geometry can be recovered from the combination of MCS curve and surface geometry.

Figure 213: PCS Curve Recovery from MCS Curve and Surface Geometry



7.2.9 JT LWPA Segment

JT LWPA Segment contains an Element that defines light weight precise analytic data for a particular part. More specifically LWPA contains the collection of analytic surfaces in the B-Rep definition of the part.

JT LWPA Segments are typically referenced by Part Node Elements (see 7.2.1.1.1.5Part Node Element) using Late Loaded Property Atom Elements (see 0Second specifies the date Second value. Valid values are [0, 59] inclusive.

Late Loaded Property Atom Element Late Loaded Property Atom ElementLate Loaded Property Atom Element). The JT LWPA Segment type supports ZLIB compression on all element data, so all elements in JT LWPA Segment use the Logical Element Header ZLIB form of element header data.

Figure 214: JT LWPA Segment data collection

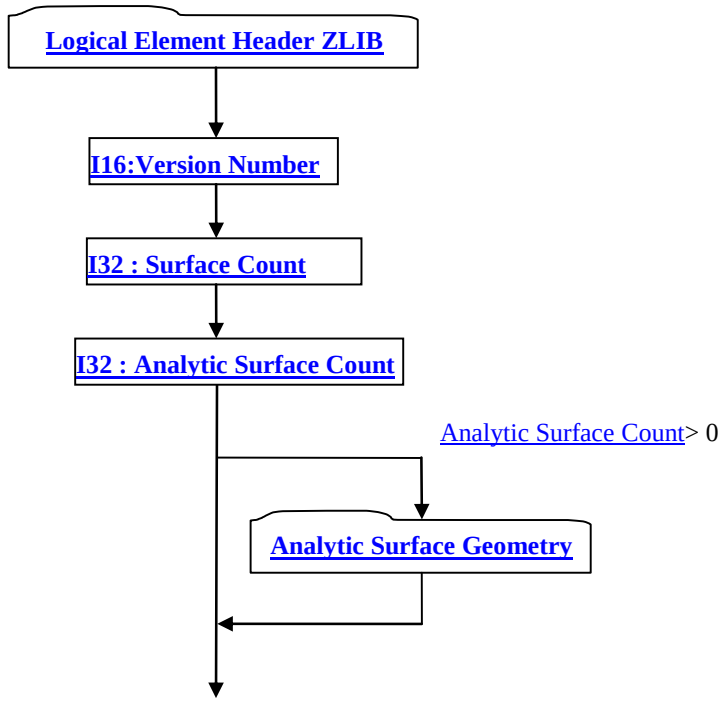
Complete description for Segment Header can be found in 7.1.3.1Segment Header.

7.2.9.1 JT LWPA Element

Object Type ID: 0xd67f8ea8, 0xf524, 0x4879, 0x92, 0x8c, 0x4c, 0x3a, 0x56, 0x1f, 0xb9, 0x3a

JT LWPA Segment UHSUHVHQWV D SDUWLFXODU 3DUW¶V SUHFLVH DQDO\WLF VXUIDFHV ,W FDQ EH YLHZHG DV D VXEVHW RI %-Rep representation where the subset refers to the complete collection of all the surfaces that are of one of the analytic types shown in the Supported Surface Type table, i.e., plane, cylinder, cone, sphere, or torus. Unlike JT B-Rep Element or XT B-Rep Element, JT LWPA Element does not contain any B-Rep topology information, nor does it contain geometric curve or point information. LWPA is designed to represent most essential part geometry information with much lighter weight on disk and much faster to load than B-Rep. Typically LWPA is less than 2 percent of B-Rep size on disk, and takes less than 5 percent time to load into memory. The analytic representation of LWPA follows Parasolid convention as detailed in Appendix F: Parasolid XT Format Reference.

Figure 215: JT LWPA Element data collection



I16:Version Number

Version Number is the version identifier for this JT LWPA Element 9HUVLRQ QXPEHUV ³ ´ LV FXUUHQW\ VXSSRUWHG

I32 : Surface Count

Surface Count indicates the number of surface entries in LWPA. The number of surface entries is equal to the number of surfaces in the B-Rep representation. The surface entry does not contain any information if the corresponding B-Rep surface is not of analytic type.

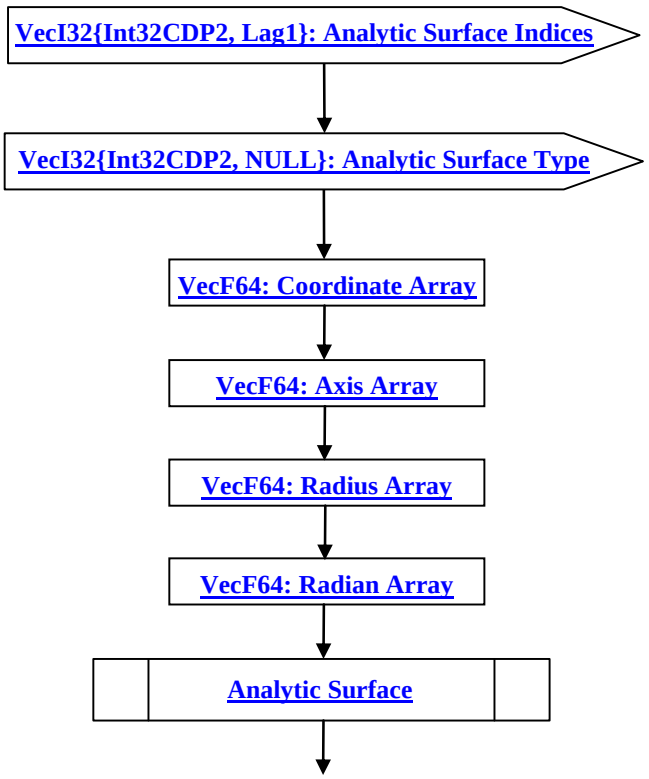
I32 : Analytic Surface Count

Analytic Surface Count indicates the number of analytic surface entries in LWPA.

7.2.9.1.1 Analytic Surface Geometry

Analytic Surface Geometry defines a collection of analytic surfaces and their mapping to the original B-Rep surfaces.

Figure 216: Analytic Surface Geometry data collection



VecI32{Int32CDP2, Lag1}: Analytic Surface Indices

Analytic Surface Indices is an integer array with its length equal to the number of analytic surfaces in the LWPA. The value of each element in this array represents the index of this analytic surface in the original B-Rep representation.

VecI32{Int32CDP2, NULL}: Analytic Surface Type

Analytic Surface Type is an integer array with its length equal to the number of analytic surfaces in the LWPA. The value of each element in this array represents the type of each analytic surface, as defined in table Supported Surface Type

VecF64: Coordinate Array

Coordinate Array contains an array of double precision floating point numbers that represent the collection of point coordinate information in the definition of the analytic surface entities. The composite type VecF64 is defined in Table 2. Each floating point number in the array is written in binary form.

VecF64: Axis Array

Axis Array contains an array of double precision floating point numbers that represent the collection of unit vector information in the definition of the analytic surface entities. The composite type VecF64 is defined in Table 2. Each floating point number in the array is written in binary form.

VecF64: Radius Array

Radius Array contains an array of double precision floating point numbers that represent the collection of radius information in the definition of the analytic surface entities. The composite type VecF64 is defined in Table 2. Each floating point number in the array is written in binary form.

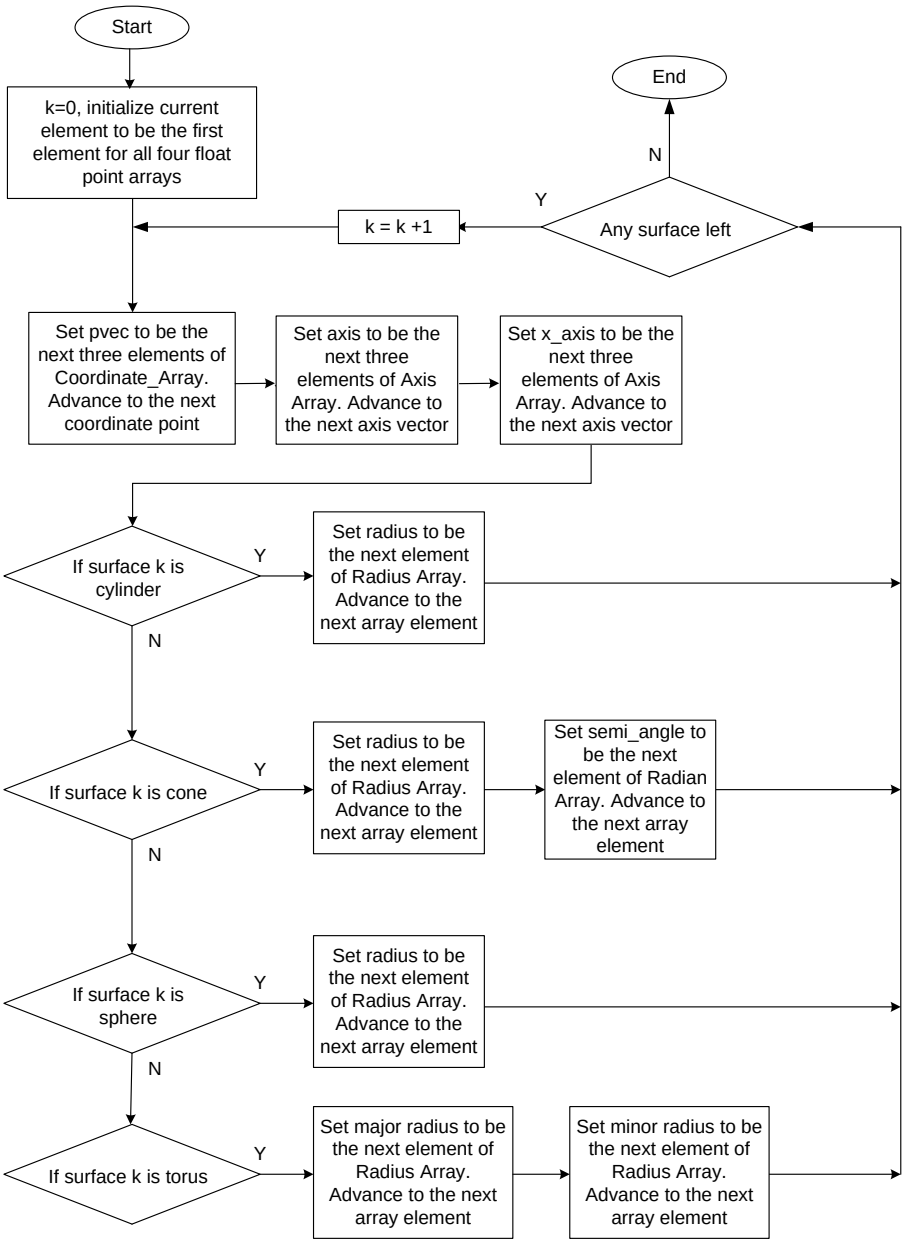
VecF64: Radian Array

Radian Array contains an array of double precision floating point numbers that represent the collection of radian information in the definition of the analytic surface entities. The composite type VecF64 is defined in Table 2. Each floating point number in the array is written in binary form.

Analytic Surface Creation

Analytic surfaces in LWPA is constructed based on the information of the above arrays, as illustrated by logical diagram in Figure 217.

Figure 217: Analytic Surface Creation



8 Data Compression and Encoding

The JT File format utilizes best-in-class compression and encoding algorithms to produce compact and efficient representations of data. The types of compression algorithms supported by the JT format vary from standard data type agnostic ZLIB deflation to advanced arithmetic algorithms that exploit knowledge of the characteristics of the data types they are compressing. Some of the JT format data collections are always stored in a compressed format, whereas other data FRQHFWLRQV VXSSRUW PXOWLSOH FRPSUHVVLRQ VWUDJH IRUPDWV WKDW TXDOLWDWLYHO\ YDU\ IURP ³/RVVOHVV´ FRPSUHVVLRQ WR more DJJUHVVLYH VWUDWHJLHV WKDW HPSOR\ ³0RVV\´ FRPSUHVVLRQ. This support by the JT format of varying qualitative levels of compression allows producers of JT data to fine tune the tradeoff between compression ratio and fidelity of the data.

In some instances, data may be encoded/compressed using multiple techniques applied on top of one another in a serial fashion (i.e. encoding applied to the output of another encoder). One common example of this multiple encoding is when an array/vector of floating point data is first quantized into some integer codes and then these resulting integer codes are further compressed/encoded using an Arithmetic or BitLength CODEC (see 8.2 Encoding Algorithms).

Beyond the data collection specific compression/encoding, some JT format Data Segment types (see 7.1.3 Data Segment) also support having a ZLIB compression conditionally applied to all the bytes of information persisted within the segment. So individual fields or collections of data may first have data type specific encoding/compression algorithms applied to them, and then if their Data Segment type supports it, the resulting data may be additionally compressed using a ZLIB deflation algorithm.

Whether, and at what qualitative level, a particular Data Se**JPHQW¶V GDWD LV FRPSUHVVHG HQFRGHG LV LQGLFDWHG WKURXJK** compression related data values stored as part of the particular Data Segment storage format. In general, aggressive application of advanced compression/encoding techniques is reserved for the heavy-weight renderable geometric data (e.g. triangles and wireframe lines) which can exist in a JT File.

The following sections document the format of the data compression/encoding within the JT file. Along with documenting the format, a technical description of the various compression/encoding algorithms is included and an example implementation of the decoding portion of the algorithms can be found within Appendix C: Decoding Algorithms ± An Implementation.

8.1 Common Compression Data Collection Formats

For convenience and brevity in documenting the JT format, this section of the reference documents the format for several FRPPRQ ³GDWD compression/HQFRGLQJ´ UHODWHG data collections that can exist in the JT format. You will find references to these common compression data collections in the 7.2 Data Segments section of the document.

8.1.1 Int32 Compressed Data Packet

The Int32 Compressed Data Packet collection represents the format used to encode/compress a collection of data into a series of Int32 based symbols. Note that the Int32 Compressed Data Packet collection can in itself contain another Int32 Compressed Data Packet collection if there aUH DQ\ ³2XW-Of-Band data ´ In the context of the JT format data compression algorithms and Int32 Compressed Data Packet ³RXW-of-EDQG GDWD´ KDV WKH IROORZLQJ PHDQLQJ

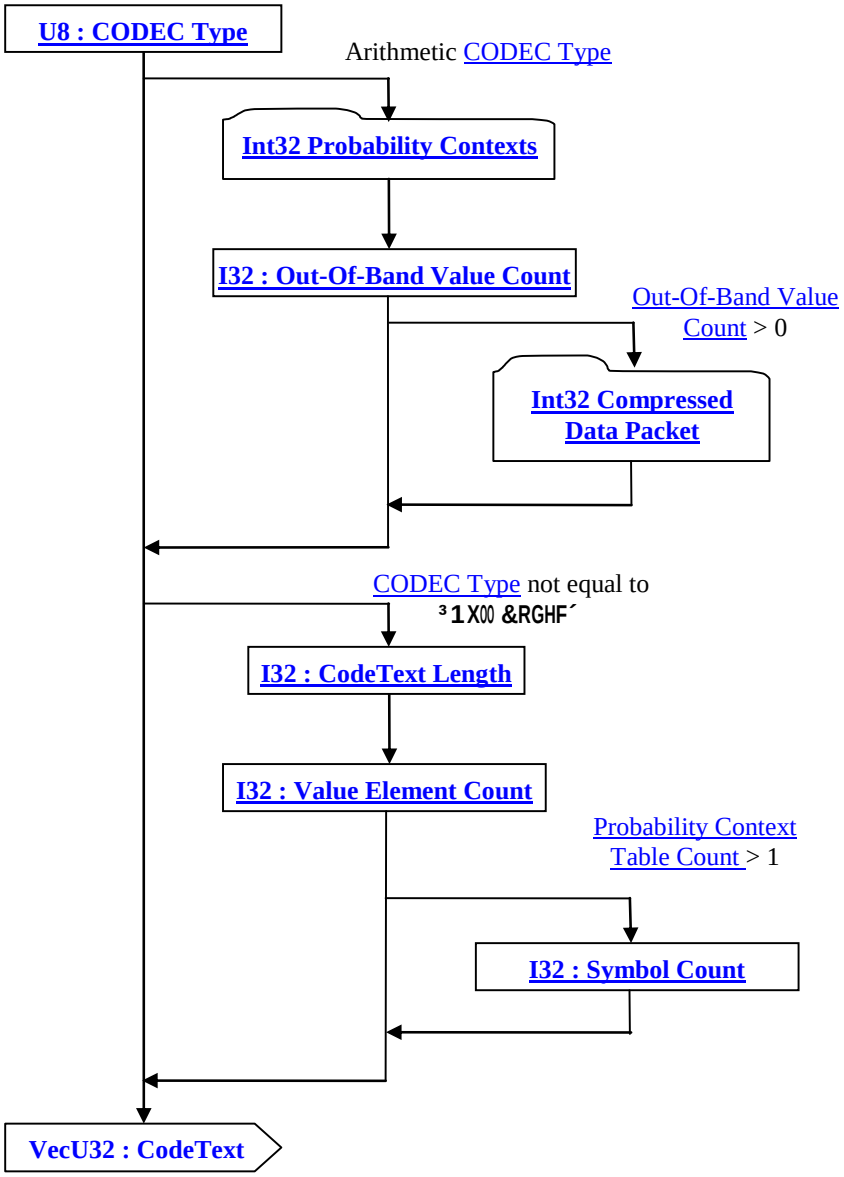
CODECs (e.g. Arithmetic, see 8.2 Encoding Algorithms for technical description) exploit the statistics present in the relative frequencies of the values being encoded. Values that occur frequently enough allow these methods to encode each of the YDOXHV DV D ³V\PER0´ LQ IHZHU ELWV WKDW LW ZRXOG WDke to encode the value itself. Values that occur too infrequently to take advantage of this property are written *aside* LQWR WKH ³RXW-of-EDQG GDWD´ DUUD\ WR EH HQFRGHG VHSDUDWH\ $Q ³HVFDSH´ V\PER0 is encoded in their place as a placeholder in the primal CODEC QRWH VHH ³6\PER0´ GDWD ILHOG GHILQLWLRQ LQ 8.1.1.1.1 Int32 Probability Context Table Entry for further GHWDLOV RQ WKH UHSUHVHQWDWLRQ RI ³HVFDSH´ V\PER0

(VVHQWLDO\ WKH ³RXW-of-EDQG GDWD´ LV WKH KLJK-entropy residue left over after the CODECs have squeezed all the advantage out of the original data stream WKDW WKH\ FDQ +RZHYHU WKLV ³RXW-of-EDQG GDWD´ LV VHQW EDFN DURXQG for another pass because sometimes there are *different* statistics to be taken advantage of. When all other coding options have been exhausted, the Bitlength CODEC is invoked. The Bitlength CODEC directly encodes all values given to it, does not require a probability contH[W DQG KHQFH QHYHU SURGXFHV DGGLWLRQDO ³RXW-of-EDQG GDWD´ 7KH E\WH VWRSV WKHUH LQ RWKHU ZRUGV

In some cases, all values may be written as "out of band" when the Codec cannot perform *any* useful compression. In this case, the encoded I32 : CodeText Length field will be 0, and the I32 : Out-Of-Band Value Count will be equal to I32 : Value

Element Count. The implied action in this case is to merely copy the Out-Of-Band value data into the output Value Element array instead of invoking the Codec.

Figure 218: Int32 Compressed Data Packet data collection



U8 : CODEC Type

CODEC Type specifies the algorithm used to encode/decode the data. See 8.2 Encoding Algorithms for complete explanation of each of the encoding algorithms.

= 0	Null CODEC
= 1	Bitlength CODEC
= 3	Arithmetic CODEC
= 4	Chopper CODEC

I32 : Out-Of-Band Value Count

Out-Of-%DQG 9DOXH &RXQW VSHFLILHV WKH QXPEHU RI YDOXHV WKDW DUH ³2XW-Of-%DQG´ This data field is only present for the Arithmetic CODEC Type.

I32 : CodeText Length

CodeText Length specifies the total number of bits of CodeText data (CodeText data field is described below). This data field is only present if CODEC Type LV QRW HTXDO WR ³1X00 &2'(&´

I32 : Value Element Count

9DOXH (OHPHQW &RXQW VSHFLILHV WKH QXPEHU RI YDOXHV WKDW WKH &2'(& LV H[SHFWHG WR GHFRGH L H LW¶V OLNH WKH ³0HQJWK´ ILHOG ZULWWHQ LI \RX¶UH WXVW ZULWLQJ RXW D YHFWRU RI LQWHJHUV 7KLV GDWD ILHOG Ls only present if CODEC Type LV QRW HTXDO WR ³1X00 &2'(&´ 8SRQ FRPSOHWLRQ RI GHFRGLQJ WKH CodeText data field below, the number of decoded Values should be equal to Value Element Count. When only a single Probability Context Table is used, Value Element Count will also be equal to the number of Symbols decoded upon completion of decoding.

I32 : Symbol Count

When two Probability Context Tables are being used, Symbol Count specifies the number of Symbols to be decoded by the Arithmetic CODEC. There is a subtlety present in the method CodecDriver::addOutputSymbol() when it is passed an Escape symbol. Only if the Codec is using Probability Context Table 0 when it receives an Escape symbol does it emit a Value from the "Out-Of-Band" data array. Because of this subtlety, the number of Symbols decoded can be larger than the number of Values produced, thus the reason for writing this field distinct from Value Element Count.

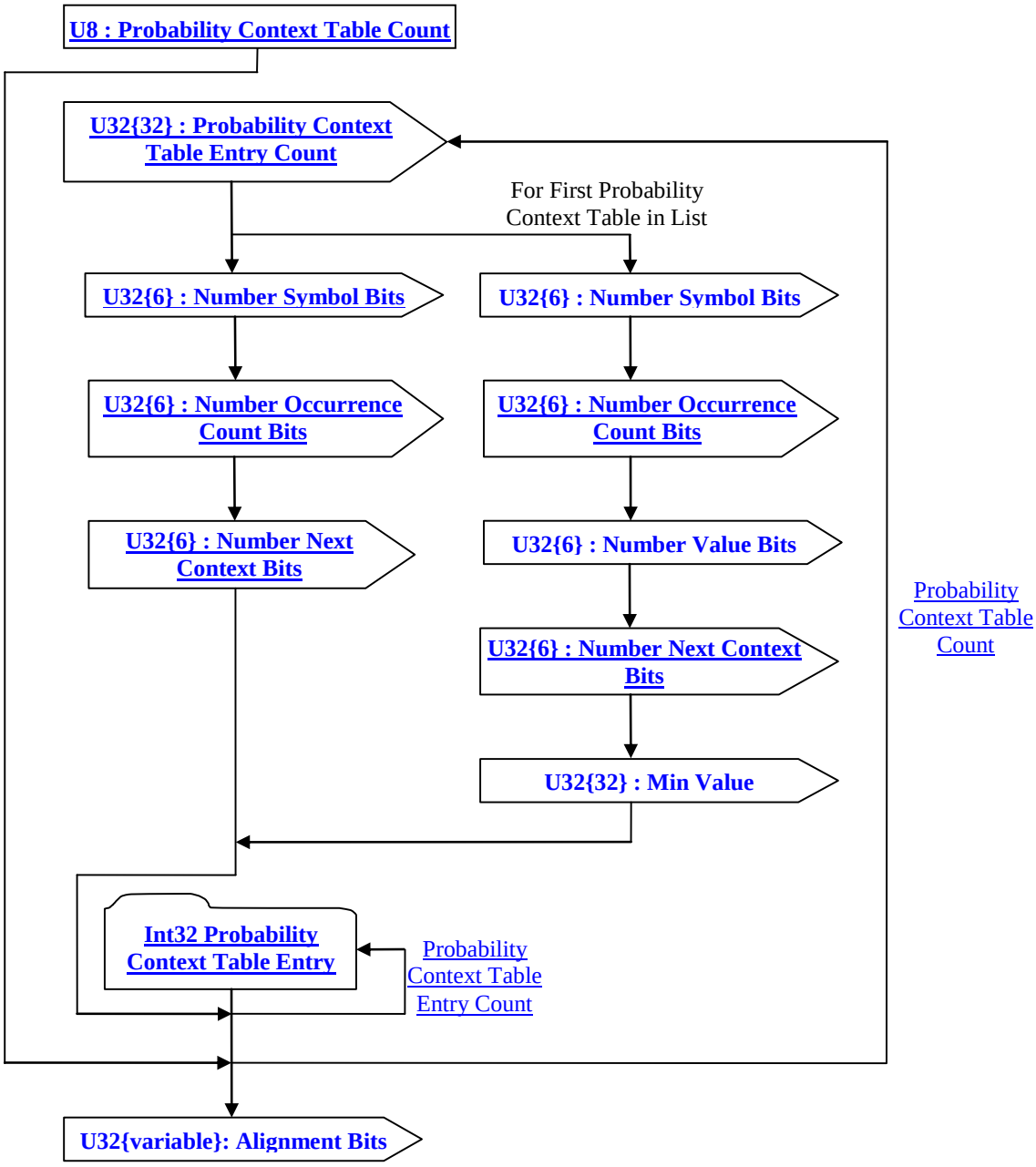
VecU32 : CodeText

CodeText is the array/vector of encoded symbols. For CODEC Type QRW HTXDO WR ³1X00 &2'(&´ WKH WRWDO QXPEHU RI ELWV RI encoded data in this array is indicated by the previously described CodeText Length data field.

8.1.1.1 Int32 Probability Contexts

Int32 Probability Contexts data collection is a list of Probability Context Tables. The Int32 Probability Contexts data collection is only present for the Arithmetic CODEC Type. A Probability Context Table is a trimmed and scaled histogram of the input values. It tallies the frequencies of the several most frequently occurring values. It is central to the operation of the arithmetic CODEC.

Figure 219: Int32 Probability Contexts data collection



U8 : Probability Context Table Count

Probability Context Table Count specifies the number of Probability Context Tables to follow and will always have a value RI HLWKHU ³ ´ RU ³ ´

U32{32} : Probability Context Table Entry Count

Probability Context Table Entry Count specifies the number of entries in this Probability Context Table.

U32{6} : Number Symbol Bits

Number Symbol Bits specifies the number of bits used to encode the Symbol range.

U32{6} : Number Occurrence Count Bits

Number Occurrence Count Bits specifies the number of bits used to encode the Occurrence Count range.

U32{6} : Number Value Bits

Number Value Bits specifies the number of bits used to encode the Associated Value range. Note that Number Value Bits is only specified in the JT file for the *first* Probability Context Table. If a second Probability Context Table is present, the Number Value Bits from the first should be used for the second as well.

U32{6} : Number Next Context Bits

Number Next Context Field Bits specifies the number of bits used for the Next Context Field in 8.1.1.1.1 Int32 Probability Context Table Entry.

U32{32} : Min Value

Min Value specifies the minimum of all Associated Values (i.e. one per table entry) stored in this Probability Context Table. This value is used to compute the real Associated Value for a Probability Context Table Entry. See Associated Value description in 8.1.1.1.1 Int32 Probability Context Table Entry.

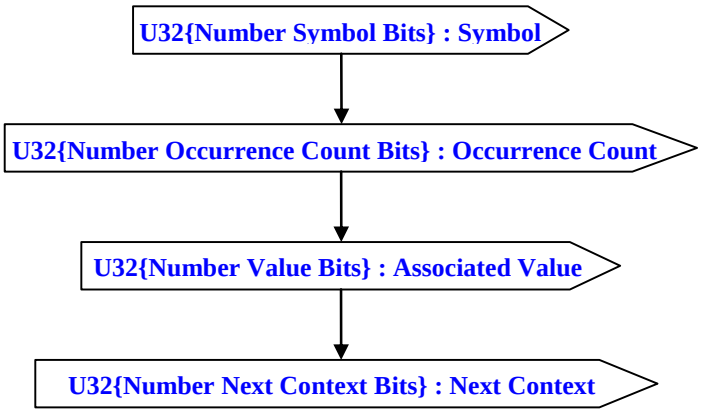
U32{variable}: Alignment Bits

Alignment Bits represents the number of additional padding bits stored to arrive at the next even multiple of 8 bits. Values of ³ ´ DUH VWRUHG LQ WKH DOLJQPHQW ELWV

Note: Data written into the JT file is always aligned on bytes. Therefore after reading in a block of bit data such as the probability context tables it is necessary to discard any remaining bits on the last byte that is read in. This is represented by WKH ³$OLJQPHQW %LWV´ HQWU\

8.1.1.1.1 Int32 Probability Context Table Entry

Figure 220: Int32 Probability Context Table Entry data collection



U32{Number Symbol Bits} : Symbol

Symbol is a small integer number associated with a specific value in the context table. It serves only to impose an order on the entries in the Probability Context Table 7KH V\PER0 LV VWRUHG ZLWK D ³ ´ DGGHG WR WKH YD0XH DQG WKXV D UHDGHU PXVW VXEWUDFW ³ ´ IURP WKH UHDG YD0XH WR JHW WKH WUXH V\PER0 YD0XH &RPSOHWH GHVFULSWLRQ IRU 1XPEHU 6\PER0 %LWV FDn be found in 8.1.1.1 Int32 Probability Contexts.

Note: Even though the symbol is written as a U32{Number Symbol Bits} it is possible to end up with a negative number after VXEWUDFWLQJ ³ ´ IURP WKH UHDG LQ YD0XH 2QH H[DPS0H WKDW ZL00 RFFXU IUHTXHQW\ LV WKH HVFDSH V\PER0 XVHG IRU RXW-of-band dDWD ZKLFK ZL00 KDYH WKH YD0XH ³ ´ LQ WKH ILOH KRZHYHU LW ZL00 EHFRPH ³- ´ LWV WUXH V\PER0 YD0XH DIWHU VXEWUDFWLQJ ³ ´ IURP WKH UHDG LQ ³ ´ YD0XH

U32{Number Occurrence Count Bits} : Occurrence Count

Occurrence Count specifies the relative frequency of the value. Complete description for Number Occurrence Count Bits can be found in 8.1.1.1 Int32 Probability Contexts.

Note: Occurrence Counts for all symbols are normalized (converted to a relative frequency) during the write process in order to ensure the minimum amount of bits possible is used to write them. This has several implications the reader should be aware of:

The sum of all Occurrence Counts is not guaranteed to equal the number of symbols to be decoded (see Value Element Count in section 8.1.1 for number of symbols to be decoded). During Arithmetic decoding as described in Appendix C: 3.2.

pDriver->numSymbolsToRead() ± Refers to the total number of symbols to be decoded (i.e. Value Element Count in section 8.1.1 when the number of Probability Context Tables is equal to 1, or Symbol Count when the number of Probability Context Tables is 2).

pCurrContext->totalCount() ± 5HIHUV WR WKH VXP RI WKH ³2FFXUUHQFH &RXQW´ YDOXHV IRU DOO WKH V\PERO DVVRFLDWHG with a Probability Context.

U32{Number Value Bits} : Associated Value

$VVRFLDWHG 9DOXH LV WKH YDOXH IURP WKH LQSXW GDWD WKDW WKH V\PERO UHSUHVHQWV 7KH &2'(&V GRQ¶W GLUHFW\ HQFRGH YDOXHV they encode symbols. Symbols, then, are associated with specific values, so when the CODEC decodes an array of symbols, you can reconstruct the array of values that was intended by looking up the symbols in the Probability Context Table. This YDOXH LV VWRUHG ZLWK ³0LQ 9DOXH´ VXEWUDFWHG IURP WKH YDOXH &RPSOHPHQW GHVFULSWLRQV IRU ³0LQ 9DOXH´ DQG 1XPEHU 9DOXH %LWV can be found in 8.1.1.1 Int32 Probability Contexts.

Note: The associated value for an escape symbol is undefined and therefore can be any valid U32 number.

U32{Number Next Context Bits} : Next Context

Next Context field specifies which Probability Context Table to use when decoding the next symbol. The value of this field will be greater than or equal to 0, and less than Probability Context Table Count.

8.1.2 Int32 Compressed Data Packet Mk. 2

The Int32 Compressed Data Packet Mk. 2 collection represents an enhanced form of the original Int32 Compressed Data Packet. Note that the Int32 Compressed Data Packet Mk. 2 collection can in itself contain another Int32 Compressed Data Packet Mk. 2 FRQWDLQLQJ LI WKHUH DUH DQ\ ³2XW-Of-%DQG GDWD´ ,Q WKH FRQWH[W RI WKH -7 IRUPDW GDWD FRPSUHVVLRQ DOJRULWKPV DQG Int32 Compressed Data Packet Mk. 2 ³RXW-of-EDQG GDWD´ KDV WKH PHDQLQJ GHVFULEHG EHORZ

Entropy CODECs (e.g. Arithmetic) exploit the statistics present in the relative frequencies of the values being encoded. 9DOXHV WKDW RFFXU IUHTXHQW\ HQRXJK DOORZ WKHVH PHWKRGV WR HQFRGH HDFK RI WKH YDOXHV DV D ³V\PERO´ LQ IHZHU ELWV WKDW LW would take to encode the value itself. Values that occur too infrequently to take advantage of this property are written *aside* LQWR WKH ³RXW-of-EDQG GDWD´ DUUD\ WR EH HQFRGHG VHSDUDWH\ $Q ³HVFDSH´ V\PERO LV HQFRGHG LQ WKHLU SDFN DV D SDFKR0GHU LQ the primal CODE& QRWH VHH ³6\PERO´ GDWD ILHOG GHILQLWLRQ LQ 8.1.2.1.1 Int32 Probability Context Table Entry Mk. 2 for further GHWDLOV RQ WKH UHSUHVHQWDWLRQ RI ³HVFDSH´ V\PERO

(VVHQWLDOO\ WKH ³RXW-of-EDQG GDWD´ LV WKH KLJK-entropy residue left over after the CODEC has squeezed all the advantage out of the oULJLQDO GDWD VWUHDP WKDW LW FDQ +RZHYHU WKLV ³RXW-of-EDQG GDWD´ LV VHQW EDFN DURXQG IRU DQRWKHU SDVV EHFDXVH sometimes there are *new* or *different* statistics to be exploited.

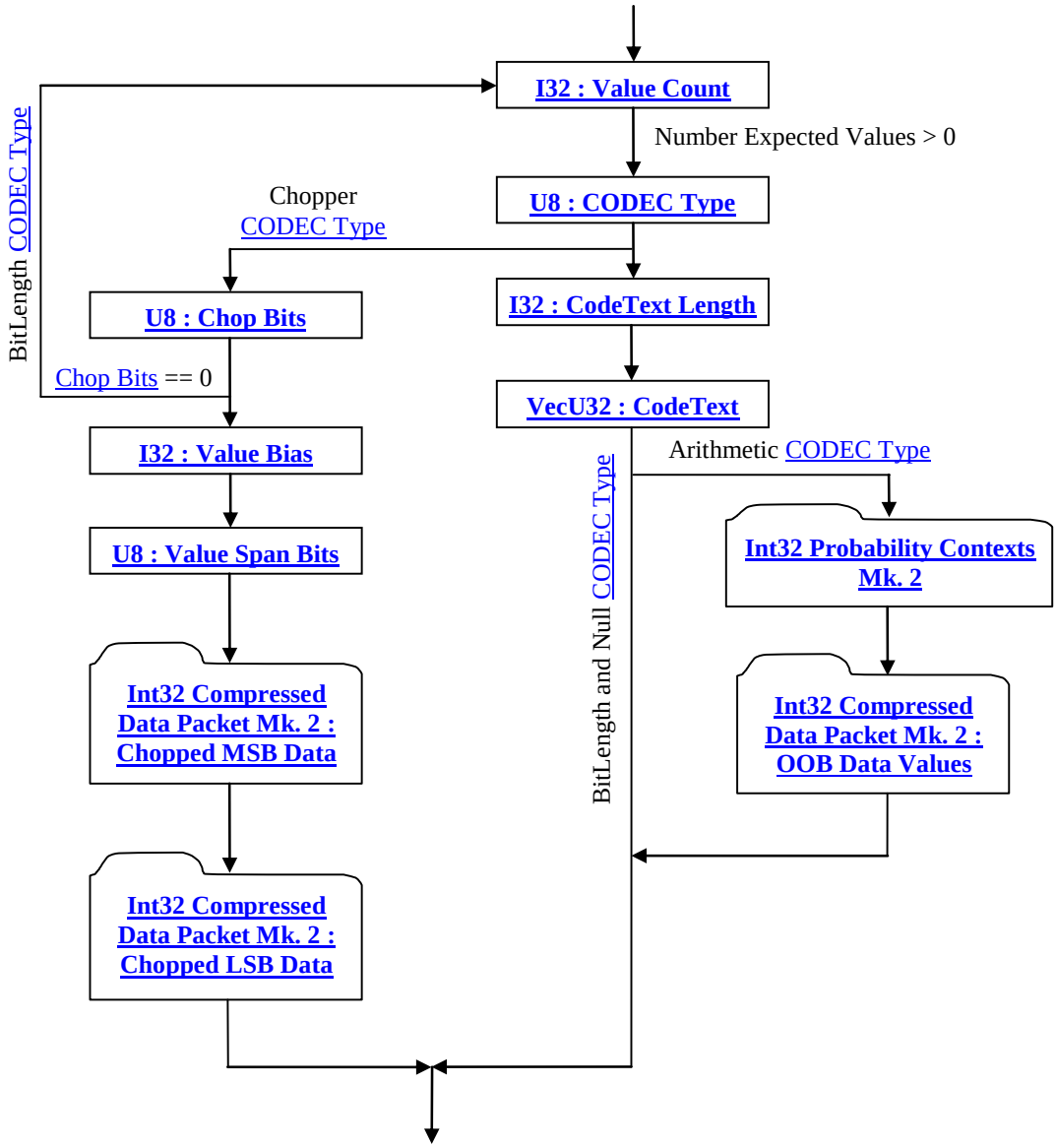
The Int32 Compressed Data Packet Mk. 2 brings the new *Chopper* pseudo-CODEC to the table. Its job is to identify fields of bits in a sequence of otherwise incompressible data that may be hiding low-entropy statistics that can be profitably exploited. In other words, it "chops" the input data up into bit fields, and then encodes them separately using the Arithmetic or BitLength CODECs, or in some cases, another round of chopping. The Chopper also removes *value bias* from the original input data array. Some input data arrays may contain values that are clustered around a certain central value. In these cases, it is profitable to first subtract out a *bias value* from the original input data. In some cases, this simple expedient may dramatically reduce the apparent field width necessary to code the variation in the original sequence.

In some cases, all values may be written as "out of band" when the Codec cannot perform *any* useful compression. In this case, the encoded I32 : CodeText Length field will be 0, and the I32 : Out-Of-Band Value Count will be equal to I32 : Value Element Count. The implied action in this case is to merely copy the Out-Of-Band value data into the output Value Element array instead of invoking the Codec.

When all other coding options have been exhausted, the Bitlength CODEC is invoked. The Bitlength CODEC directly encodes all values given to it, does not require a probability FRQWH[W DQG KHQFH QHYHU SURGXFHV DGGLWLRQDO ³RXW-of-EDQG GDWD´ The byte stops there, in other words.

Note that in the diagram below, encoding can loop back recursively for Out-Of-Band data and chopper fields. *For JT v9 files, the maximum recursion depth may not exceed three*.

Figure 221: Int32 Compressed Data Packet Mk. 2 data collection



I32 : Value Count

9DOXH &RXQW VSHFLILHV WKH QXPEHU RI YDOXHV WKDW WKH &2'(& LV H[SHFWHG WR GHFRGH L H LW¶V OLNH WKH ³OHQJWK´ ILHOG ZULWWHQ LQ \RX¶UH MXVW ZULWLQJ RXW D YHFWRU RI LQWHJHUV 8SRQ FRPSOHWLRQ RI GHFRGLQJ WKH CodeText data field below, the number of decoded Values should be equal to Value Count. When only a single Probability Context Table is used, Value Element Count will also be equal to the number of Symbols decoded upon completion of decoding.

U8 : CODEC Type

CODEC Type specifies the algorithm used to encode/decode the data. See 8.2 Encoding Algorithms for complete explanation of each of the encoding algorithms.

= 0	Null CODEC
= 1	Bitlength CODEC
= 3	Arithmetic CODEC
= 4	Chopper CODEC

I32 : CodeText Length

CodeText Length specifies the total number of bits of CodeText data (CodeText data field is described below).

VecU32 : CodeText

CodeText is the array/vector of encoded symbols. For CODEC Type QRW HTXDO WR ³1X00 &2'(&´ WKH WRWDO QXPEHU RI ELWV RI encoded data in this array is indicated by the previously described CodeText Length data field.

U8 : Chop Bits

Chop Bits specifies the number of high-order bits "chopped off" from the *biased* input data array and coded separately from the low-order bits. Repeated applications of the Chopper pseudo-CODEC can expose low-entropy bit fields that would be inaccessible by directly coding the data array. Chop Bits is the number of bits coded into the Chopped MSB Data field.

I32 : Value Bias

Value Bias is the (signed) number that is subtracted from the original input data array elements *before* computing Value Span Bits and Chop Bits. See Chopped LSB Data below for a full explanation of how to reconstitute the original data values using Value Bias and the two chopped fields.

U8 : Value Span Bits

Value Span Bits specifies the total bit width of the *biased* input data array. Note that Value Span Bits minus Chop Bits is the number of low-order bits present in the Chopped LSB Data field.

Int32 Compressed Data Packet Mk. 2 : Chopped MSB Data

This field contains the separately compressed most significant bits of the *biased* input data array, whose elements contain Value Span Bits bits of significance. In other words, this field contains the bit field from the *biased* data array beginning at bit number ValueSpan-ChopBits and ending at bit number ValueSpan-1 inclusive. This field may contain negative numbers.

Int32 Compressed Data Packet Mk. 2 : Chopped LSB Data

This field contains the separately compressed most significant bits of the original input data array, whose elements contain Value Span Bits bits of significance. In other words, this field contains the bit field from the original data array beginning at bit number 0 and ending at bit number ValueSpan-ChopBits-1 inclusive. This field may only contain positive numbers; all bits above this range must encode to 0. A pseudo-code representation of the re-constituting the original data values is as follows:

```
OrigValue[i] = (LSBValue[i] | (MSBValue[i] << (ValSpanBits - ChopBits))) + ValueBias;
```

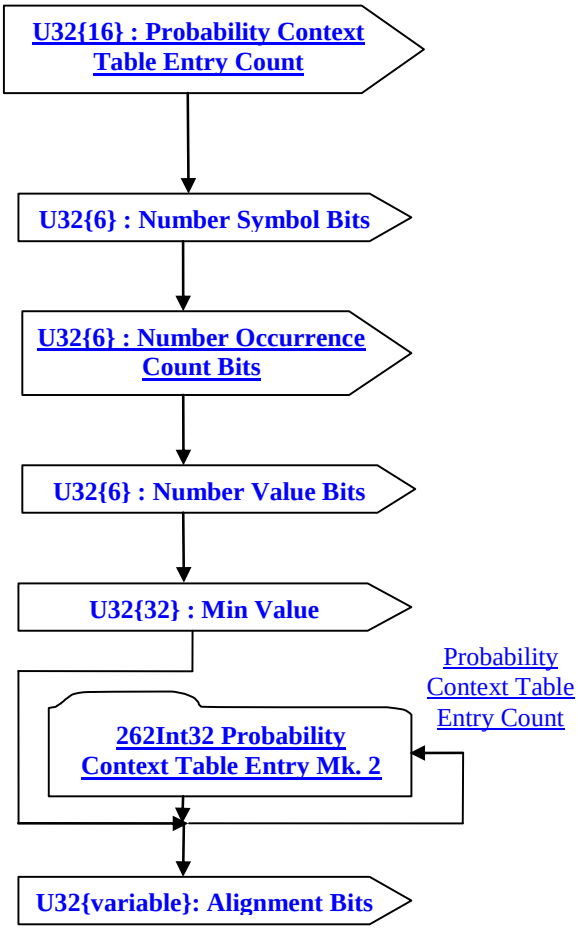
Int32 Compressed Data Packet Mk. 2 : OOB Data Values

This field encodes the out-of-band values associated with the Arithmetic CODEC.

8.1.2.1 Int32 Probability Contexts Mk. 2

Int32 Probability Contexts Mk. 2 data collection encodes a Probability Context Table, and is present only for the Arithmetic CODEC Type. A Probability Context Table is a trimmed and scaled histogram of the input values. It tallies the frequencies of the several most frequently occurring values. It is central to the operation of the Arithmetic CODEC.

Figure 222: Int32 Probability Contexts Mk. 2 data collection



U32{16} : Probability Context Table Entry Count

Probability Context Table Entry Count specifies the number of entries in this Probability Context Table.

U32{6} : Number Symbol Bits

Number Symbol Bits specifies the number of bits used to encode the Symbol range.

U32{6} : Number Occurrence Count Bits

Number Occurrence Count Bits specifies the number of bits used to encode the Occurrence Count range.

U32{6} : Number Value Bits

Number Value Bits specifies the number of bits used to encode the Associated Value range. Note that Number Value Bits is only specified in the JT file for the *first* Probability Context Table. If a second Probability Context Table is present, the Number Value Bits from the first should be used for the second as well.

U32{32} : Min Value

Min Value specifies the minimum of all Associated Values (i.e. one per table entry) stored in this Probability Context Table. This value is used to compute the real Associated Value for a Probability Context Table Entry. See Associated Value description in 8.1.1.1.1 Int32 Probability Context Table Entry.

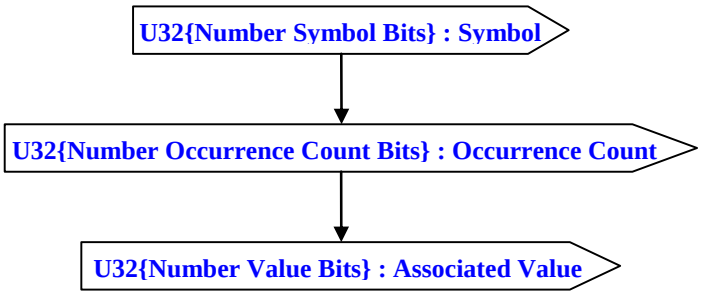
U32{variable}: Alignment Bits

Alignment Bits represents the number of additional padding bits stored to arrive at the next even multiple of 8 bits. Values of ³ ´ DUe stored in the alignment bits.

Note: Data written into a JT file is always aligned on bytes. Therefore after reading in a block of bit data such as the probability context tables it is necessary to discard any remaining bits on the last byte that is read in. This is represented by WKH ³$OLJQPHQW %LWV´ HQWU\

8.1.2.1.1 Int32 Probability Context Table Entry Mk. 2

Figure 223: Int32 Probability Context Table Entry Mk. 2 data collection



U32{Number Symbol Bits} : Symbol

Symbol is a small integer number associated with a specific value in the context table. It serves only to impose an order on WKH HQWULHV LQ WKH 3UREDELOLW\ &RQWH[W 7DEOH 7KH V\PERO LV VWRUHG ZLWK D ³ ´ DGGHG WR WKH YDOXH DQG WKXV D UHDGHU PXVW VXEWUDFW ³ ´ IURP WKH UHDG YDOXH WR JHW WKH WUXH V\PERO YDOXH &RPSOHWH GHVFULSWLRQ IRU 1XPEHU Symbol Bits can be found in 8.1.2.1 Int32 Probability Contexts Mk. 2.

Note: Even though the symbol is written as a U32{Number Symbol Bits} it is possible to end up with a negative number after VXEWUDFWLQJ ³ ´ IURP WKH UHDG LQ YDOXH 2QH H[DPSOH WKDW ZLOO RFFXU IUHTXHQW\ LV WKH HVFDSH V\PERO XVHG IRU RXW-of-band data whiFK ZLOO KDYH WKH YDOXH ³ ´ LQ WKH ILOH KRZHYHU LW ZLOO EHFRPH ³- ´ LWV WUXH V\PERO YDOXH DIWHU VXEWUDFWLQJ ³ ´ IURP WKH UHDG LQ ³ ´ YDOXH

U32{Number Occurrence Count Bits} : Occurrence Count

Occurrence Count specifies the relative frequency of the value. Complete description for Number Occurrence Count Bits can be found in 8.1.2.1 Int32 Probability Contexts Mk. 2.

Note: Occurrence Counts for all symbols are normalized (converted to a relative frequency) during the write process in order to ensure the minimum amount of bits possible is used to write them while closely approximating their actual frequency. This has several implications the reader should be aware of:

The sum of all Occurrence Counts is not guaranteed to equal the number of symbols to be decoded (see I32 : Value Count in section 8.1.2 for number of symbols to be decoded).

During Arithmetic decoding as described in Appendix C: 3.2.

pDriver->numSymbolsToRead() ± Refers to the total number of symbols to be decoded (i.e. I32 : Value Count in section 8.1.2).

pCurrContext->totalCount() ± 5HIHUV WR WKH VXP RI WKH ³2FFXUUHQFH &RXQW´ YDOXHV IRU DOO WKH V\PERO DVVRFLDWed with a Probability Context.

U32{Number Value Bits} : Associated Value

AssociaWHG 9DOXH LV WKH YDOXH IURP WKH LQSXW GDWD WKDW WKH V\PERO UHSUHVHQWV 7KH &2'(&V GRQ¶W GLUHFW\ HQFRGH YDOXHV they encode symbols. Symbols, then, are associated with specific values, so when the CODEC decodes an array of symbols, you can reconstruct the array of values that was intended by looking up the symbols in the Probability Context Table. This YDOXH LV VWRUHG ZLWK ³0LQ 9DOXH´ VXEWUDFWHG IURP WKH YDOXH &RPSOHWH GHVFULSWLRQV IRU ³0LQ 9DOXH´ DQG 1XPEHU 9DOXH %LWV can be found in 8.1.2.1 Int32 Probability Contexts Mk. 2.

Note: The associated value for an escape symbol is undefined and therefore can be any valid U32 number.

8.1.3 Float64 Compressed Data Packet

The Float64 Compressed Data Packet collection represents the format used to encode/compress a collection of data into a series of Float64 based symbols. This compression format also uses WKH FRQFHSW RI ³RXW-of-EDQG GDWD´ LQ LWV GDWD FRQWHQWV definition. In the context of the JT format data compression algorithms and Float64 Compressed Data Packet ³RXW-of-band GDWD´ KDV WKH IRORZLQJ PHDQLQJ

The Arithmetic CODEC (see 8.2 Encoding Algorithms for technical description) can exploit the statistics present in the relative frequencies of the values being encoded. Values that occur frequently enough allow the CODEC to encode each of WKH YDOXHV DV D ³V\PER0´ LQ IHZHU ELWV WKDW LW ZRX0G WDNH WR HQFRGH WKH YDOXH LWVH0I 9DOXHV WKDW RFFXU WRR LQIUHTXHQW\ WR take advantage of this property are written *aside* LQWR WKH ³RXW-of-EDQG GDWD´ DUUD\ $Q ³HVFDSH´ V\PER0 L H YDOXH RI ³-2´ is encoded in their place as a marker in the primal CODEC. Essentially the ³RXW-of-EDQG GDWD´ is the high-entropy junk/residue/slag left over after the CODECs have squeezed all the advantage out that it can.

Whereas the Int32 Compressed Data Packet (see 8.1.1 Int32 Compressed Data Packet WKHQ VHQGV WKLV ³RXW-of-EDQG GDWD´ back around through a new CODEC looking for *different* statistics to be taken advantage of, the Float64 Compressed Data Packet VLPS0\ ZULWHV RXW WKH ³RXW-of-EDQG GDWD´ DUUD\ ZLWK QR DGGLWLRQD0 HQFRGLQJ DWWHPPSWHG

In some cases, all values may be written as "out of band" when the Codec cannot perform *any* useful compression. In this case, the encoded I32 : CodeText Length field will be 0, and the I32 : Out-Of-Band Value Count will be equal to I32 : Value Element Count. The implied action in this case is to merely copy the Out-Of-Band value data into the output Value Element array instead of invoking the Codec.

U8 : CODEC Type

CODEC Type specifies the algorithm used to encode/decode the data. See 8.2 Encoding Algorithms for complete explanation of each of the encoding algorithms.

= 0	Null CODEC
= 1	Bitlength CODEC
= 3	Arithmetic CODEC
= 4	Chopper CODEC

F64 : Value Range Min

Value Range Min specifies the minimum of the value range used to encode the values. This data field is only present if CODEC Type LV QRW HTXDO WR ³1X00 &2'(&´

F64 : Value Range Max

Value Range Max specifies the maximum of the value range used to encode the values. This data field is only present if CODEC Type LV QRW HTXDO WR ³1X00 &2'(&´

I32 : Out-Of-Band Value Count

Out-Of-Band Value Count specifies the number of values that are ³2XW-Of-%DQG´ This data field is only present if CODEC Type LV QRW HTXDO WR ³1X00 &2'(&´

VecF64 : Out-Of-Band Values

Out-Of-Band Values specifies the vector OLVW RI ³2XW-Of-%DQG´ YDOXHV This data field is only present if CODEC Type is not HTXDO WR ³1X00 &2'(&´

I32 : CodeText Length

CodeText Length specifies the total number of bits of CodeText data (described below). This data field is only present if CODEC Type LV QRW HTXDO WR ³1X00 &2'(&´

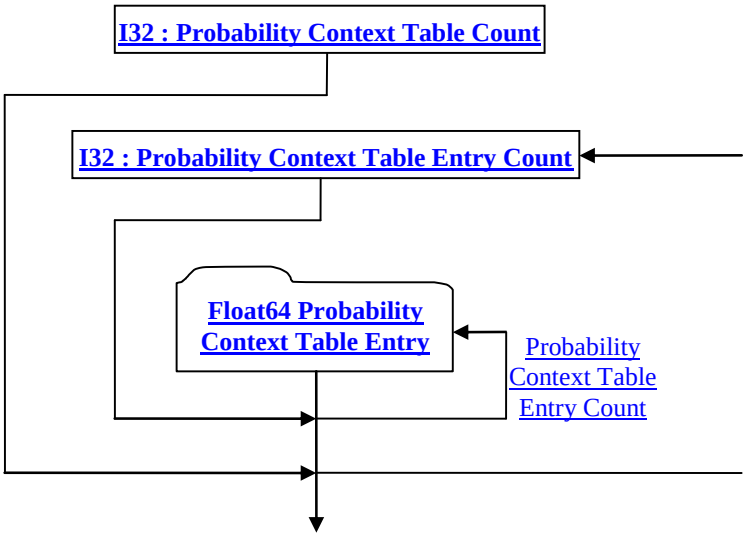
I32 : Value Element Count

Value Element Count specifies WKH QXPEHU RI YDOXHV WKDW WKH &2'(& LV H[SHFWHG WR GHFRGH L H LW¶V OLNH WKH ³0HQJWK´ ILHOG ZULWWHQ LI \RX¶UH WXVW ZULWLQJ RXW D YHFWRU RI LQWHJHUV . This data field is only present if CODEC Type LV QRW HTXDO WR ³1X00 &2'(&´ 8SRQ FRPSOHWLRQ RI GHFRGLQJ WKH CodeText data field below, the number of decoded symbol values should be equal to Value Element Count.

I32 : Symbol Count

When two Probability Context Tables are being used, Symbol Count specifies the number of Symbols to be decoded by the Arithmetic CODEC. There is a subtlety present in the method CodecDriver::addOutputSymbol() when it is passed an Escape symbol. Only if the Codec is using Probability Context Table 0 when it receives an Escape symbol does it emit a Value from the6'Out-Of-Band" data array. Because of this subtlety, the number of Symbols decoded can be larger than the number of OHFaP

Figure 225: Float64 Probability Contexts data collection



I32 : Probability Context Table Count

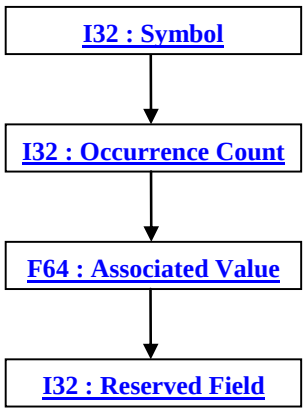
Probability Context Table Count specifies the number of Probability Context Tables to follow and will always have a value RI HLWKHU ³ ´ RU ³ ´.

I32 : Probability Context Table Entry Count

Probability Context Table Entry Count specifies the number of entries in this Probability Context Table.

8.1.3.1.1 Float64 Probability Context Table Entry

Figure 226: Float64 Probability Context Table Entry data collection



I32 : Symbol

Symbol is a small integer number associated with a specific value in the context table. It serves only to impose an order on the entries in the Probability Context Table. 1RWH WKDW D YDOXH RI ³- ´ UHSUHVHQWV WKH ³HVFDSH´ V\PERO SODFHKROGHU HQFRGHG IRU ³RXW-of-EDQG GDWD´ VHH 8.1.3 Float64 Compressed Data Packet for additional details).

I32 : Occurrence Count

Occurrence Count specifies the relative frequency of the value.

F64 : Associated Value

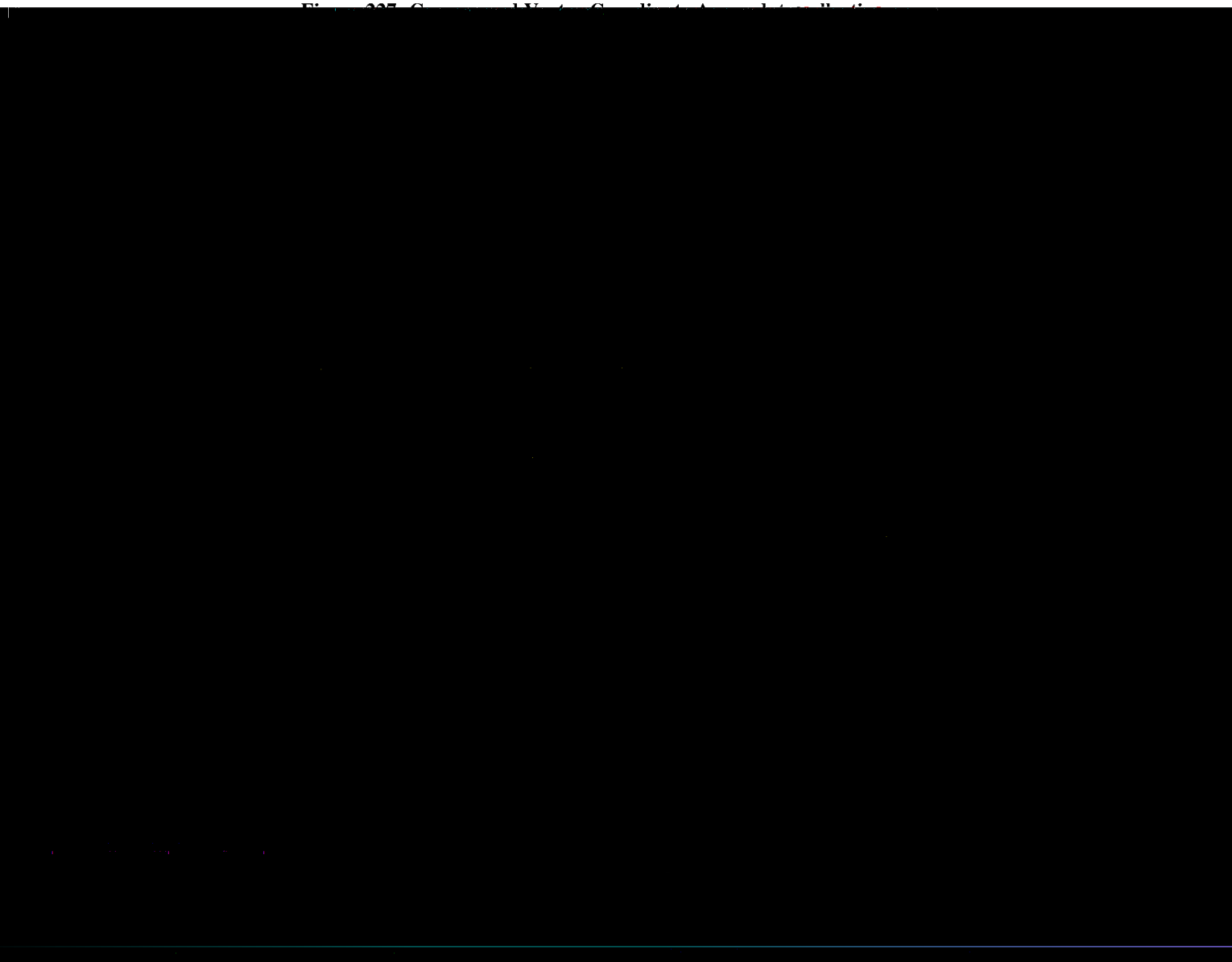
Associated Value is the value (from the input data) that the symbol represents. The CODECs don't directly encode *values,* they encode *symbols*. Symbols, then, are associated with specific values, so when the CODEC decodes an array of symbols, you can reconstruct the array of values that was intended by looking up the symbols in the Probability Context Table.

I32 : Reserved Field

Reserved Field is a data field reserved for future JT format expansion.

8.1.4 Compressed Vertex Coordinate Array

The Compressed Vertex Coordinate Array data collection contains the quantization data/representation for a set of vertex coordinates.



VecU32{Int32CDP2, Lag1} : Vertex Coord Exponents

Vertex Coord Exponents is a vector of Floating Point Exponents and Sign for all the ith component values of a set of vertex coordinates. Vertex Coord Exponents uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2, Lag1} : Vertex Coord Mantissae

Vertex Coord Mantissae is a vector of Floating Point Mantissae for all the ith component values of a set of vertex coordinates. Vertex Coord Mantissae uses the Int32 version of the CODEC to compress and encode data.

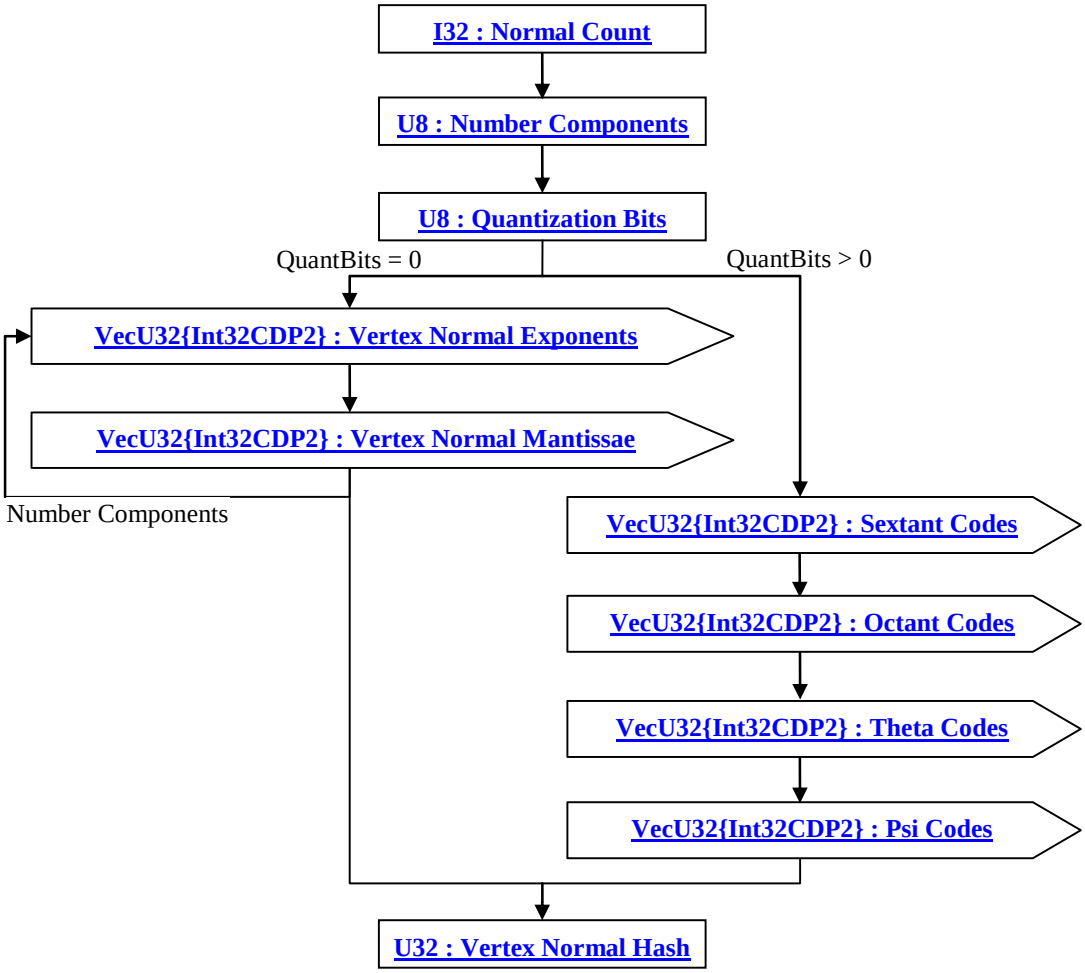
VecU32{Int32CDP2, Lag1} : Vertex Coord Codes

9HUWH[&RRUG &RGHV LV D YHFWRU RI TXDQWL]HU ³FRGHV´ IRU DOO the ith component values of a set of vertex coordinates. Vertex Coord Codes uses the Int32 version of the CODEC to compress and encode data.

I32 : Vertex Coordinate Hash

The Vertex Coordinate Hash is the combined hash of the unique vertex coordinate records. If the number of quantiz ber

Figure 228: Compressed Vertex Normal Array data collection



I32 : Normal Count

Normal count specifies the number of normals. This number should equal the total number of vertex records.

U8 : Number Components

Number Components specifies the number of normal components present for each vertex record in the set of vertex records.

U8 : Quantization Bits

The number of bits used when the Deering Normal CODEC if quantization is enabled. A value of 0 denotes that quantization is disabled.

VecU32{Int32CDP2} : Vertex Normal Exponents

Vertex Normal Components is a vector of Floating Point Exponents for all the ith component values of a set of vertex coordinates. Vertex Normal Components uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2} : Vertex Normal Mantissae

Vertex Normal Components is a vector of Floating Point Mantissae for all the ith component values of a set of vertex coordinates. Vertex Normal Components uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2} : Sextant Codes

Sextant Codes is a vector RI ³FRGHV´ RQH SHU QRUPDO IRU D VHW RI QRUPDOV LGHQWLI\LQJ ZKLFK 6H[WDQW RI WKH FRUUHVSRQGLQJ sphere Octant each normal is located in. Sextant Codes uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2} : Octant Codes

2FWDQW &RGHV LV D YHFWRU RI ³FRGHV´ RQH SHU QRUPDO IRU D VHW RI QRUPDOV LGHQWLI\LQJ ZKLFK VSKHUH 2FWDQW HDFK QRUPDO LV located in. Octant Codes uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2} : Theta Codes

Theta Codes LV D YHFWRU RI ³FRGHV´ RQH SHU QRUPDO IRU D VHW RI QRUPDOV UHSUHVHQWLQJ LQ 6H[WDQW FRRUGLQDWHV WKH TXDQWL]HG WKHWD DQJOH IRU HDFK QRUPDO¶V ORFDWLRQ RQ WKH XQLW UDGLXV VSKHUH ZKHUH WKHWD DQJOH LV GHILQHG DV WKH DQJOH LQ VSKHULFDO coordinates about the Y-axis on a unit radius sphere. Theta Codes uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2} : Psi Codes

3VL &RGHV LV D YHFWRU RI ³FRGHV´ RQH SHU QRUPDO IRU D VHW RI QRUPDOV UHSUHVHQWLQJ LQ 6H[WDQW FRRUGLQDWHV WKH TXDQWL]ed Psi DQJOH IRU HDFK QRUPDO¶V ORFDWLRQ RQ WKH XQLW UDGLXV VSKHUH ZKHUH 3VL DQJOH LV GHILQHG DV WKH ORQJLWXGLQDO DQJOH LQ VSKHULFDO coordinates from the y = 0 plane on the unit radius sphere. Psi Codes uses the Int32 version of the CODEC to compress and encode data

U32 : Vertex Normal Hash

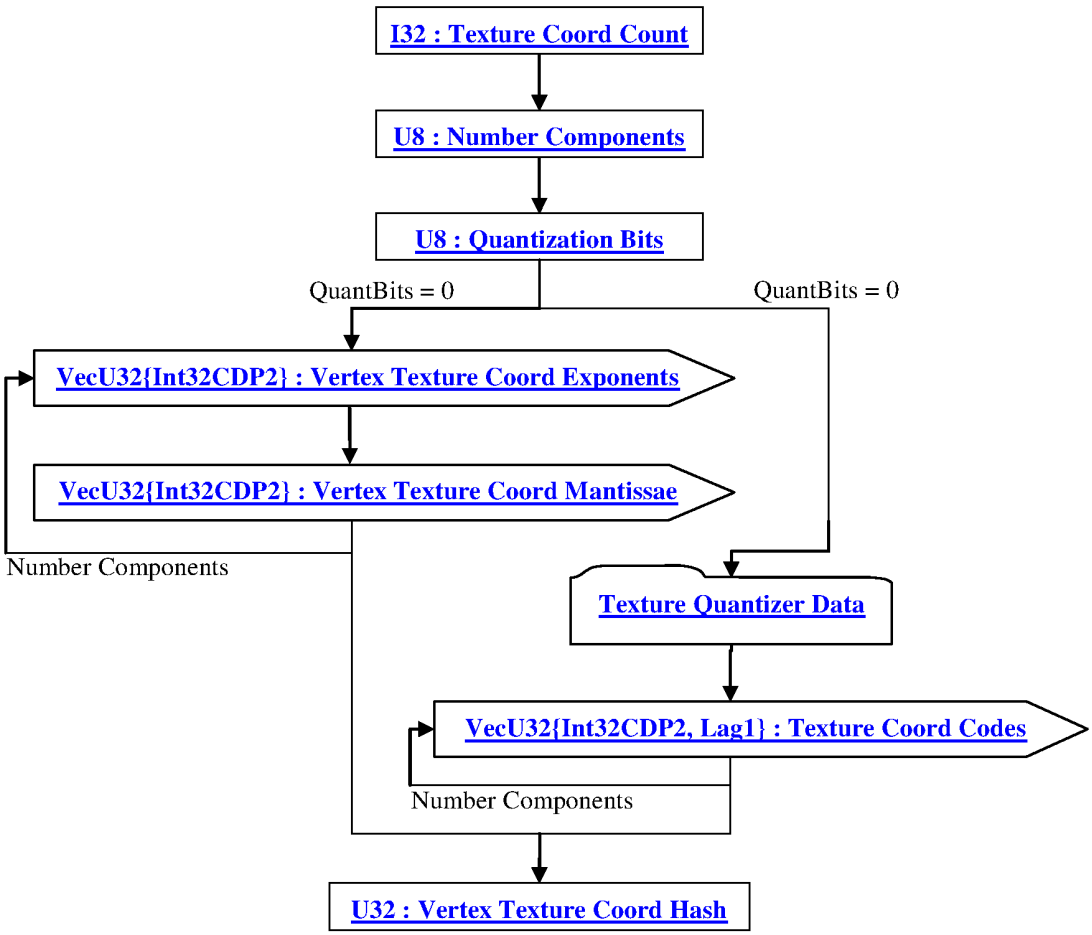
The Vertex Normal Hash is the combined hash of the vertex normals. If the number of quantization bits is equal to zero the hash value is equal to the combined hash of the vertex normal values for each of the component arrays. If the number of quantization bits is greater than 0 the hash value is equal to the combined hash of the Sextant, Octant, Theta, and Psi Codes for all vertex records. Refer to section 9.5 for a more detailed description on hashing.

```
UInt32 uHash   = 0;
uInt32 nVtxRec = 0;
vecF32 vNorm[nVtxRec][3];
vecU32 vSextant, vOctant. vTheta, vPsi;
...
if ( uQuantBits == 0 ) {
  for ( int i=0 ; i<nComp ; i++ ) {
    for ( int j=0 ; j<nVtxRec ; j++) {
      uHash = hash32( (UInt32*)(&vNorm[j][i]), 1, uHash );
    }
  }
} else {
    uHash = hash32( &vSextant, nVtxRec, uHash );
    uHash = hash32( &vOctant, nVtxRec, uHash );
    uHash = hash32( &vTheta, nVtxRec, uHash );
    uHash = hash32( &vPsi, nVtxRec, uHash );
}
```

8.1.6 Compressed Vertex Texture Coordinate Array

The Compressed Vertex Texture Coordinate Array data collection contains the quantization data/representation for a set of vertex texture coordinates. Compressed Vertex Texture Coordinate Array data collection is only present if previously read vertex bindings denote texture coordinates are presents (See Vertex Shape LOD Data U64 : Vertex Bindings for complete explanation of the vertex bindings).

Figure 229: Compressed Vertex Texture Coordinate Array data collection



Complete description for Texture Quantizer Data can be found in 8.1.10 Texture Quantizer Data.

I32 : Texture Coord Count

Color count specifies the number of Texture Coordinates. This number should equal the total number of vertex records.

U8 : Number Components

Number Components specifies the number of Texture Coordinate components present for each vertex record in the set of vertex records.

U8 : Quantization Bits

Number of Bits specifies the quantized size (i.e. the number of bits of precision) for each of the components. The actual number of quantization bits used is specified within Texture Quantizer Data. Value must be within range [0:24] inclusive.

VecU32{Int32CDP2} : Vertex Texture Coord Mantissae

Vertex Texture Coordinate Components is a vector of Floating Point Mantissae for all the ith component values of a set of vertex coordinates. Vertex Texture Coordinate Components uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2, Lag1} : Texture Coord Codes

V-7H[WXUH &RRUG &RGHV LV D YHFWRU RI TXDQWL]HU ³FRGHV´ IRU DOO WKH QWK-component of a set of vertex texture coordinates. V-Texture Coord Codes uses the Int32 version of the CODEC to compress and encode data.

U32 : Vertex Texture Coord Hash

The Vertex Texture Coord Hash is the combined hash of the Vertex Texture Coordinates. If the number of quantization bits is equal to zero the hash value is equal to the combined hash of the vertex texture coordinate values for each of the component arrays. If the number of quantization bits is greater than 0 the hash value is equal to the combined hash of the vertex texture coordinates codes for each of the component arrays. Refer to section 9.5 for a more detailed description on hashing.

```
UInt32 uHash    = 0;
uInt32 nVtxRec = 0;
vecF32 vTexCoord[nVtxRec][4];
vecU32 vCodes[4];
...
if ( uQuantBits == 0 ) {
  for ( int i=0 ; i<nComp ; i++ ) {
    for ( int j=0 ; j<nVtxRec ; j++) {
      uHash = hash32( (UInt32*)(&vTexCoord[j][i]), 1, uHash );
    }
  }
} else {
  for ( int i=0 ; i<nComp ; i++ ) {
    uHash = hash32( &vCodes[i], nVtxRec, uHash );
  }
}
```

8.1.7 Compressed Vertex Color Array

The Compressed Vertex Color Array data collection contains the quantization data/representation for a set of vertex colors. Compressed Vertex Color Array data collection is only present if previously read Color Binding value is not equal to zero (See Vertex Shape LOD Data for complete explanation of Color Binding data field).

Figure 230: Compressed Vertex Color Array data collection

VecU32{Int32CDP2} : Vertex Color Exponents

Vertex Normal Components is a vector of Floating Point Exponents for all the ith component values of a set of vertex coordinates. Vertex Normal Components uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2} : Vertex Color Mantissae

Vertex Normal Components is a vector of Floating Point Mantissae for all the ith component values of a set of vertex coordinates. Vertex Normal Components uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2, Lag1} : Hue/Red Codes

+XH 5HG &RGHV LV D YHFWRU RI TXDQWL]HU ³FRGHV´ IRU DOO WKH +XH 5HG FRORU FRPSRQHQWV RI D VHW RI YHUWH[FRORUV +XH 5HG Codes uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2, Lag1} : Sat/Green Codes

6DW *UHHQ &RGHV LV D YHFWRU RI TXDQWL]HU ³FRGHV´ IRU DOO WKH 6DWXUDWLRQ *UHHQ FRORU FRPSRQHQWV RI D VHW RI YHUWH[FRORUV Sat/Green Codes uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2, Lag1} : Value/Blue Codes

Value/B0XH &RGHV LV D YHFWRU RI TXDQWL]HU ³FRGHV´ IRU DOO WKH 9DOXH %0XH FRORU FRPSRQHQWV RI D VHW RI YHUWH[FRORUV Value/Blue Codes uses the Int32 version of the CODEC to compress and encode data.

VecU32{Int32CDP2, Lag1} : Alpha Codes

Alpha Codes is a vector RI TXDQWL]HU ³FRGHV´ IRU DOO WKH $0SKD FRORU FRPSRQHQWV RI D VHW RI YHUWH[FRORUV $0SKD &RGHV XVHV the Int32 version of the CODEC to compress and encode data.

U32 : Vertex Color Hash

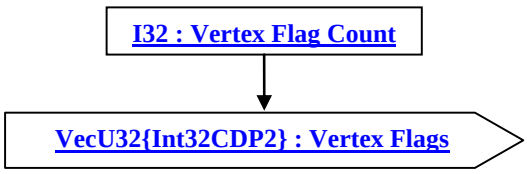
The Vertex Color Hash is the combined hash of the vertex colors. If the number of quantization bits is equal to zero the hash value is equal to the combined hash of the vertex color values for each of the component arrays. If the number of quantization bits is greater than 0 the hash value is equal to the combined hash of the Hue/Red, Sat/Green, Value/Blue, and Alpha Codes for all vertex records. Refer to section 9.5 for a more detailed description on hashing.

```
UInt32 uHash    = 0;
uInt32 nVtxRec = 0;
vecF32 vCol[nVtxRec][3];
vecU32 vHue, vSat, vVal, vAlp;
...
if ( uQuantBits == 0 ) {
  for ( int i=0 ; i<nComp ; i++ ) {
    for ( int j=0 ; j<nVtxRec ; j++) {
      uHash = hash32( (UInt32*)(&vCol[j][i]), nVtxRec, uHash );
    }
  }
} else {
    uHash = hash32( &vHue, nVtxRec, uHash );
    uHash = hash32( &vSat, nVtxRec, uHash );
    uHash = hash32( &vVal, nVtxRec, uHash );
    uHash = hash32( &vAlp, nVtxRec, uHash );
}
```

8.1.8 Compressed Vertex Flag Array

The Compressed Vertex Flag Array data collection contains the quantization data/representation for per vertex flags. Compressed Vertex Flag Array data collection is only present if previously read Vertex Flag Binding value is not equal to zero.

Figure 231: Compressed Vertex Flag Array data collection



I32 : Vertex Flag Count

Vertex flag count specifies the number of vertex flags. This number should be equal to the total number of vertex records.

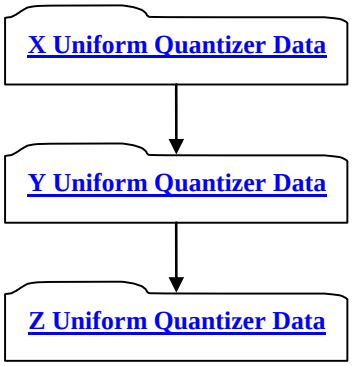
VecU32{Int32CDP2} : Vertex Flags

Vertex Flags is a vector of per vertex bit flags encoded as integers with valid values of either 0 (false) or 1 (true). Vertex Flags uses the Int32 version of the CODEC to compress and encode data.

8.1.9 Point Quantizer Data

A Point Quantizer Data collection is made up of three Uniform Quantizer Data collections; there is a separate Uniform Quantizer Data collection for the X, Y, and Z values of point coordinates.

Figure 232: Point Quantizer Data collection

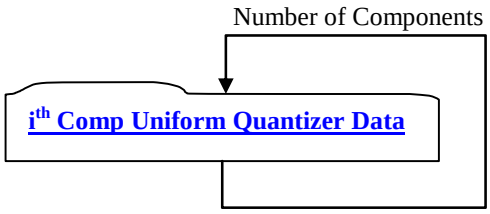


Complete description for X Uniform Quantizer Data, Y Uniform Quantizer Data and Z Uniform Quantizer Data can be found in 8.1.12 Uniform Quantizer Data.

8.1.10 Texture Quantizer Data

A Texture Quantizer Data collection is made up of n Uniform Quantizer Data collections; there is a separate Uniform Quantizer Data collection for each component of the texture coordinates. The number of components is not specified within the quantizer, but rather is determined by the number of texture components present in the underlying data (See Compressed Vertex Texture Coordinate Arrays U8 : Number Components).

Figure 233: Texture Quantizer Data collection



Complete description for U Uniform Quantizer Data, and V Uniform Quantizer Data can be found in 8.1.12 Uniform Quantizer Data.

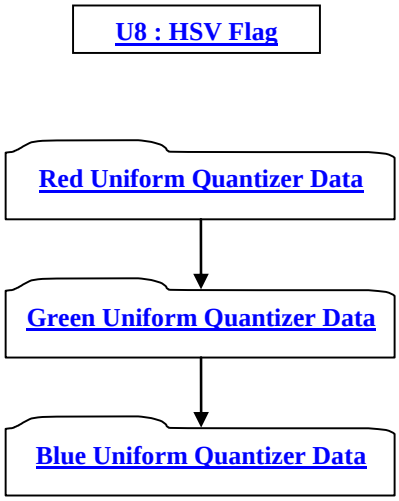
8.1.11 Color Quantizer Data

A Color Quantizer Data collection contains the quantizer information for each of the color components. The Color Quantizer utilizes a separate Uniform Quantizer Data collection for each of the 4 color components, but if the HSV color model is being used, then it is not necessary to store a complete Uniform Quantizer Data Collection.

For the HSV model, since the range values for each color component are constant, only the Number of Bits of precision for each color FRPSRQHQW¶V Uniform Quantizer is stored. The Uniform Quantizer range values for the HSV color components should always be assumed to be the following:

Component	Quantizer Range	
	Min	Max
Hue	0.0	6.0
Saturation	0.0	1.0
Value	0.0	1.0
Alpha	0.0	1.0

Figure 234: Color Quantizer Data collection



Complete descriptions for Red Uniform Quantizer Data, Green Uniform Quantizer Data, Blue Uniform Quantizer Data, and Alpha Uniform Quantizer Data can be found in 8.1.12 Uniform Quantizer Data. These four Uniform Quantizer Data collections are only present when data field HSV Flag = = 0.

U8 : HSV Flag

HSV Flag is a flag indicating whether color component data is stored in HSV color model form.

= 0	Color component data stored in RGB color model form.
= 1	Color component data stored in HSV color model form.

U8 : Number of Hue Bits

Number of Hue Bits specifies the quantized size (i.e. the number of bits of precision) for the Hue component of the color. Number of Hue Bits data is only present when data field HSV Flag = = 1.

U8 : Number of Saturation Bits

Number of Saturation Bits specifies the quantized size (i.e. the number of bits of precision) for the Saturation component of the color. Number of Saturation Bits data is only present when data field HSV Flag = = 1.

U8 : Number of Value Bits

Number of Value Bits specifies the quantized size (i.e. the number of bits of precision) for the Value component of the color. Number of Value Bits data is only present when data field HSV Flag = = 1.

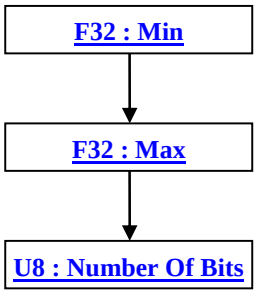
U8 : Number of Alpha Bits

Number of Alpha Bits specifies the quantized size (i.e. the number of bits of precision) for the Alpha component of the color. Number of Alpha Bits data is only present when data field HSV Flag = = 1.

8.1.12 Uniform Quantizer Data

The Uniform Quantizer Data collection contains information that defines a scalar quantizer/dequantizer (encoder/decoder) whose range is divided into levels of equal spacing.

Figure 235: Uniform Quantizer Data collection



F32 : Min

Min specifies the minimum of the quantized range.

F32 : Max

Max specifies the maximum of the quantized range.

U8 : Number Of Bits

Number of Bits specifies the quantized size (i.e. the number of bits of precision). In general, this value must satisfy the IRQRZLQJ FRQGLWLRQ ³ 1XPEHU 2I %LWV ´

8.1.13 Compressed Entity List for Non-Trivial Knot Vector

Compressed Entity List for Non-Trivial Knot Vector data collection specifies index identifiers (i.e. indices to particular entities within a list of entities) for a set of entities that contain Non-Trivial Knot Vectors. The entity types which can contain non-trivial knot vectors include:

JT B-Rep NURBS Surfaces

JT B-Rep PCS NURBS Curves

JT B-Rep MCS NURBS Curves

Wireframe MCS NURBS Curves

Note that any one occurrence of Compressed Entity List for Non-Trivial Knot Vector data collection will only contain index identifiers for one particular type of the above listed entities. The entity type is inferred based on the data collection which includes/references the Compressed Entity List for Non-Trivial Knot Vector.

A trivial knot vector is one which completely satisfies all conditions of at least one of the following cases:

Case-1 for trivial knot vector

Number of knots is an even number

Knot vector has a [0:1] knot range

There are no interior knots (i.e. NumberKnots = = 2 * (NurbsEntityDegree + 1)

Case-2 for trivial knot vector

Number of knots is an even number.

Knot vector has a [0:1] knot range

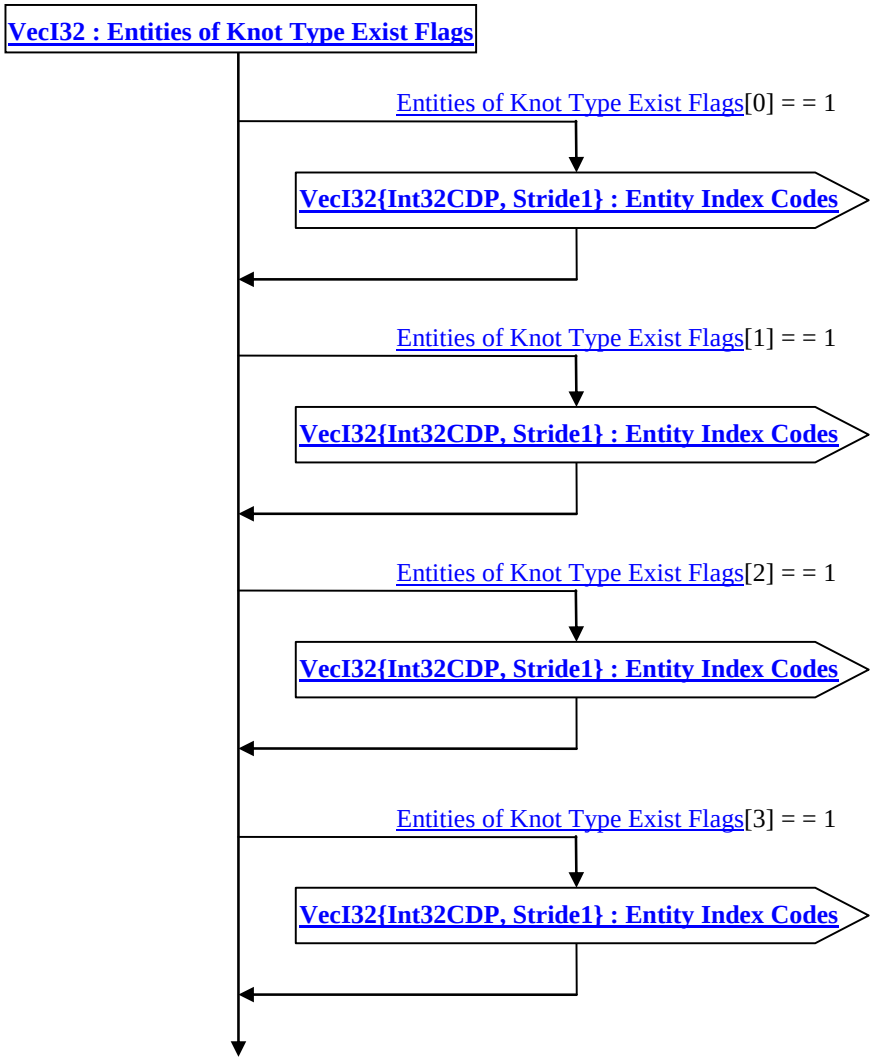
NurbsEntityDegree < 3

Difference between successive non-repeating knots (i.e. KnotDelta) is:

KnotDelta = 2.0 / (NumberKnots ± (2.0 * NurbsEntityDegree))

Any knot vector which does not satisfy one of the above cases is a non-trivial knot vector.

Figure 236: Compressed Entity List for Non-Trivial Knot Vector data collection



VecI32 : Entities of Knot Type Exist Flags

Entities of Knot Type Exist Flags, is a vector of flags indicating for each knot vector type whether Entity Index ID data collections exist/follow for that knot vector type. Knot Vectors are categorized into types based on the following characteristics: whether internal knots occur in *adjacent pairs* and whether the knot range is [0:1] or some other $[x_1:x_2]$ range.

Currently there are four knot vector types, so this Entities of Knot Type Exist Flags vector should be of length four. The four flags have the following meaning:

[0]	Flag indicating whether Entity IDs data collection exists for ³(YHQ &RXQW> @ 5DQJH´ NQRW type. Knots in this category have their knot range on [0:1], internal knots occur in *adjacent pairs*, *except* when there are no internal knots, in which case Type = 1 instead. = 0 **±** No Entity IDs data collection exists. = 1 **±** Entity IDs data collection exists.
[1]	Flag indicating whether Entity IDs data collection exists IRU ³(YHQ &RXQW>[x_1:x_2@ 5DQJH´ knot type. Knots in this category have their knot range on $[x_1:x_2]$, and internal knots occur in *adjacent pairs*. = 0 **±** No Entity IDs data collection exists. = 1 **±** Entity IDs data collection exists.
[2]	Flag indicating whether Entity IDs data collection exists IRU ³2GG &RXQW> @ 5DQJH´ NQRW type. Knots of this type have their knot range on [0:1], and are not Type 0. = 0 **±** No Entity IDs data collection exists. = 1 **±** Entity IDs data collection exists.
[3]	Flag indicating whether Entity IDs data collection exists IRU ³2GG &RXQW>[x_1:x_2@ 5DQJH´ NQRW type. Knots of this type have their knot range on $[x_1:x_2]$, and are not Type 1. = 0 **±** No Entity IDs data collection exists. = 1 **±** Entity IDs data collection exists.

Examples of knot vectors of Type 0:

```
0 0 X X 1 1
0 0 X X Y Y 1 1
0 0 X X Y Y Z Z 1 1
```

Examples of knot vectors of Type 1:

```
0 0 1 1         (Note: This is the exception to Type 0)
X X Y Y
X X Y Y Z Z
X X Y Y Z Z W W
```

Examples of knot vectors of Type 2:

```
0 0 X 1 1
0 0 X Y 1 1
0 0 X Y Z 1 1
0 0 X X X 1 1
0 0 X X Y Z Z 1 1
```

Examples of knot vectors of Type 3:

```
X X Y Z Z
X X Y Z W W
```

With this information in hand, the reader is able to reconstruct complete knot vectors in the following manner. When reconstructing the knot vector, you only take just enough values from the decoded knot value array. This may be as few as one. All the other values are inferred. Here's a sketch of the reconstruction algorithm:

```
// Number of knots in the knot vector
cNumKnots = numCtlPts + degree + 1;
// Necessary knot multiplicity at both ends of the knot vector
cClamping = degree + 1;
switch (knotType) {
```

```
    // Clamping is 0..1, internal knots occur in ADJACENT PAIRS
    // *EXCEPT* when there are no internal knots, in which case
    // Type = 1 instead.
    case 0: numVals = (cNumKnots - 2 * cClamping)/2;
    // Clamping is X1..X2, internal knots occur in ADJACENT PAIRS
    case 1: numVals = (cNumKnots - 2 * cClamping)/2 + 2;
    // Clamping is 0..1, and not Type 0
    case 2: numVals = (cNumKnots - 2 * cClamping);
    // Clamping is X1..X2, and not Type 1
    case 3: numVals = (cNumKnots - 2 * cClamping) + 2;
}
// numVals is the number of non-inferrable knot values needed
// Let vVals be the knot vector value array
// vKnot will be the final output knot vector
if (knotType is either 0 or 2)
    Set vKnot[0 .. cClamping-1] to 0
    Set vKnot[cNumKnots-cClamping .. cNumKnots-1] to 1
else
    Set vKnot[0 .. cClamping-1] to vVals[0]
    Set vKnot[cNumKnots-cClamping .. cNumKnots-1] to vVals[numVals-1]
Set vKnot[cClamping .. cNumKnots-cClamping-1] from vVals[1 .. numVals-2]
```

Vecl32{Int32CDP, Stride1} : Entity Index Codes

(QWLW\ ,QGH[&RGHV LV D YHFWRU RI TXDQWL]HU ³FRGHV´ UHSUHVHQWLQJ HQWLW\ LQGH[LGHQWLILHUV IRU D VHW RI HQWLWLHV L H LQGLFHV to particular entities within a list of entities). Entity Index Codes uses the Int32 version of the CODEC to compress and encode data.

8.1.14 Compressed Control Point Weights Data

Compressed Control Point Weights Data collection is the compressed and/or encoded representation of weight data for some set of Control Points. All NURBS based geometry use this data collection to compress/encode Control Point Weight data.

Figure 237: Compressed Control Point Weights Data collection

I32 : Weights Count

Weights Count specifies the total number of Weights. This count can differ from the Control Point count (see 7.2.3.1.4.1.3 NURBS Surface Control Point Counts) because if the Control Point Dimensionality is non-rational (see data field NURBS Surface Control Point Dimensionality in 7.2.3.1.4.1 Surfaces Geometric Data), then no Weight fi2

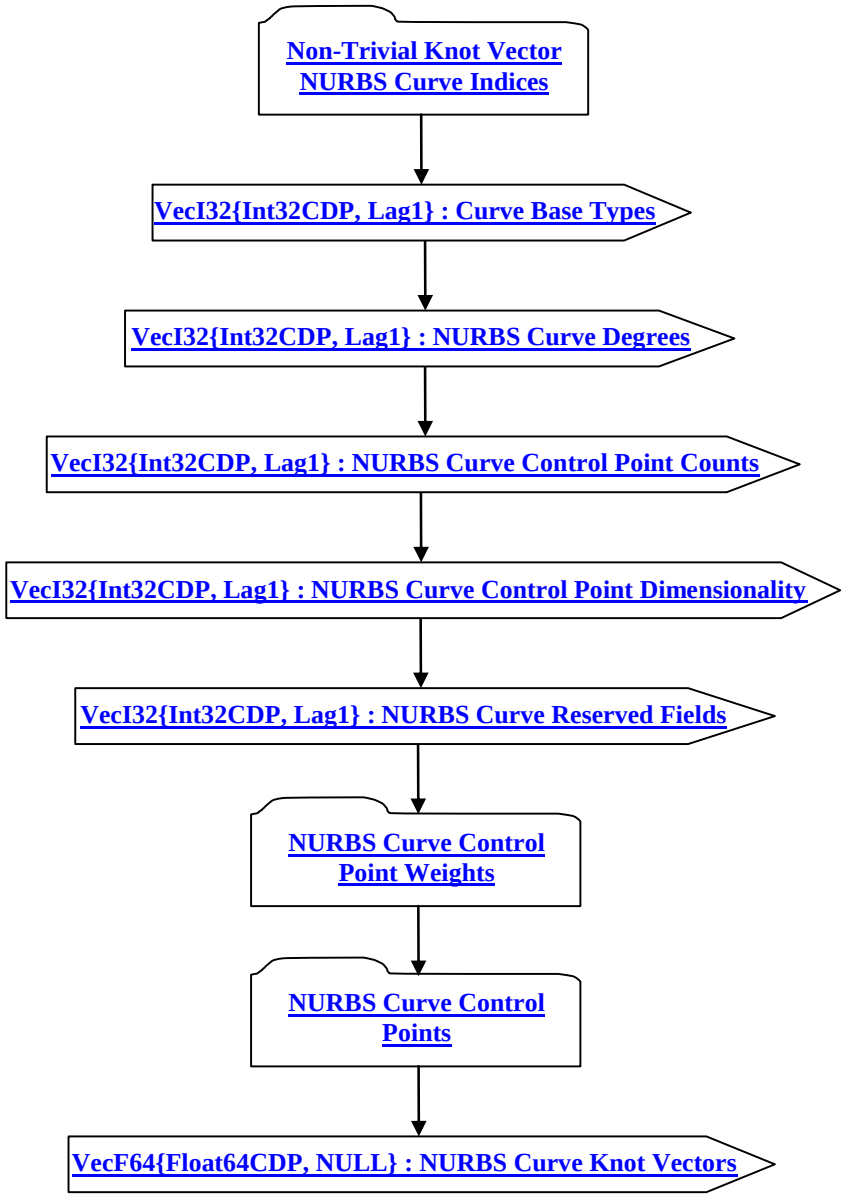
VecF64{Float64CDP, NULL} : Weight Values

Weight Values is a vector of weight values for the conditional set of weights. Weight Values uses the Float64 version of the CODEC to compress and encode data.

8.1.15 Compressed Curve Data

Compressed Curve Data collection contains JT B-Rep or Wireframe Rep compressed/encoded geometric Curve data. Currently only NURBS Curve types are supported as part of this data collection. Complete documentation for JT B-Rep and Wireframe Rep can be found in sections 7.2.3.1 JT B-Rep Element and 7.2.5.1 Wireframe Rep Element respectively.

Figure 238: Compressed Curve Data collection



VecI32{Int32CDP, Lag1} : Curve Base Types

Each Curve is assigned a base type identifier. Curve Base Types is a vector of base type identifiers for each Curve in a list of Curves. Currently only NURBS Curve Base Type is supported, but a type identifier is still included in the specification to allow for future expansion of the JT Format to support other curve types.

In an uncompressed/decoded form the Curves base type identifier values have the following meaning:

= 1	Curve is a NURBS curve

Curve Base Types uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : NURBS Curve Degrees

NURBS Curve Degrees is a vector of Curve degree values for each NURBS Curve in a list of Curves (there is a stored value for each NURBS Curve in the list). NURBS Curve Degrees uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : NURBS Curve Control Point Counts

NURBS Curve Control Point Counts is a vector of control point counts for each NURBS Curve in a list of curves (there is a stored value for each NURBS Curve in the list). NURBS Curve Control Point Counts uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : NURBS Curve Control Point Dimensionality

NURBS Curve Control Point Dimensionality is a vector of control point dimensionality values for each NURBS Curve in a list of Curve s(i.e. there is a stored values for each NURBS Curve in the list).

In an uncompressed/decoded form the control point dimensionality values meaning is dependent upon the NURBS Entity type.

For NURBS UV Curve entities the dimensionality value has the following definition:

= 2	Non-Rational (each control point has 2 coordinates)
= 3	Rational (each control point has 3 coordinates)

For NURBS XYZ Curve entities the dimensionality value has the following definition:

= 3	Non-Rational (each control point has 3 coordinates)
= 4	Rational (each control point has 4 coordinates)

NURBS Curve Control Point Dimensionality uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP, Lag1} : NURBS Curve Reserved Fields

NURBS Curve Reserved Fields is a vector of data reserved for future expansion of the JT format. Each NURBS Curve in a list of Curves has one reserved data field entry in this NURBS Curve Reserved Fields vector. NURBS Curve Reserved Fields uses the Int32 version of the CODEC to compress and encode data

VecF64{Float64CDP, NULL} : NURBS Curve Knot Vectors

NURBS Curve Knot Vectors is a list of knot vector values for each NURBS Curve having non-trivial knot vectors in a list of Curves (i.e. there are stored values for each non-trivial knot vector NURBS Curve in the list). All these NURBS Curve non-trivial knot vectors are accumulated into this single list in the same order as the Curve appears in the Curve list (i.e. Curve-N Non-Trivial Knot Vector, Curve-M Non-Trivial Knot Vector, etc.). The NURBS Curves for which knot vectors are stored (i.e. those containing non-trivial knot vectors) are identified in data collection Non-Trivial Knot Vector NURBS Curve

Indices documented in 8.1.15.1 Non-Trivial Knot Vector NURBS Curve Indices. NURBS Curve Knot Vectors uses the Float64 version of the CODEC to compress and encode data.

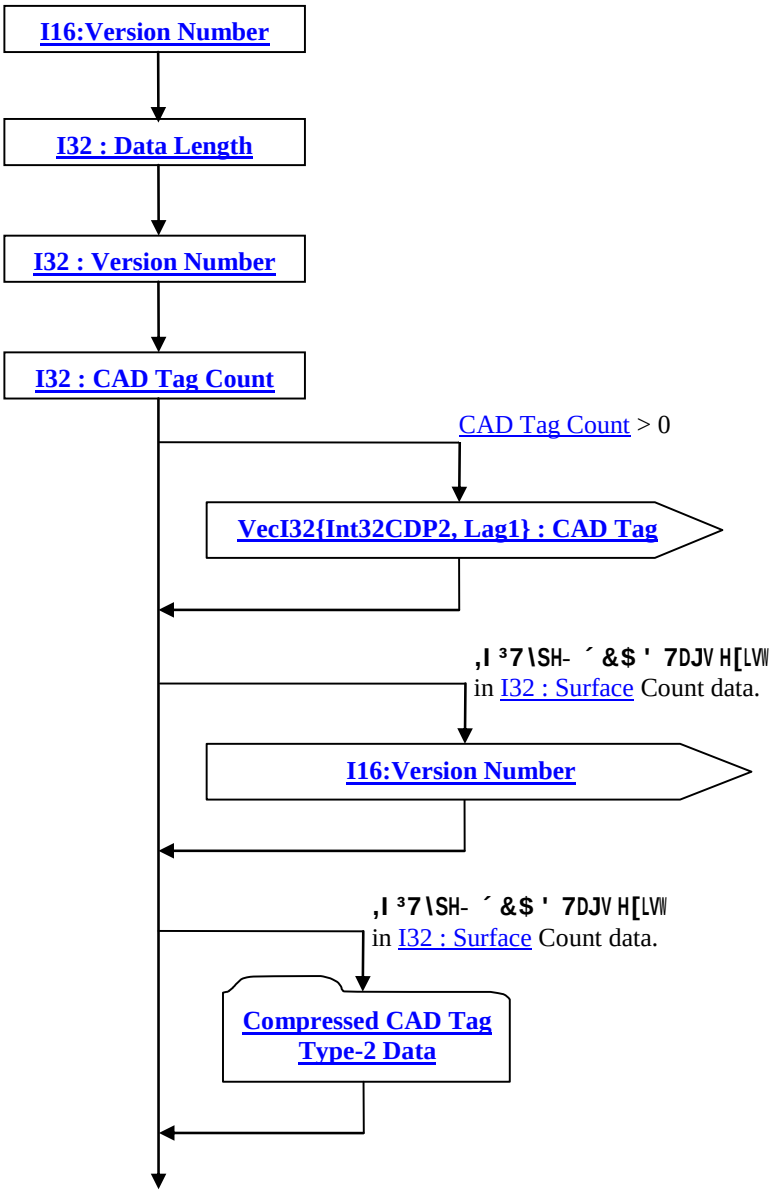
8.1.15.1 Non-Trivial Knot Vector NURBS Curve

8.1.16 Compressed CAD Tag Data

The Compressed CAD Tag Data collection contains the persistent IDs, as defined in the CAD System, to uniquely identify individual CAD entities (e.g. Faces and Edges of a JT B-Rep, PMI, etc.). Exactly what CAD entity types have CAD Tags and what order they are stored in Compressed CAD Tag Data is defined by users of this data collection (e.g. 7.2.3.1.6 B-Rep CAD Tag Data, 7.2.6.2.7 PMI CAD Tag Data)

What constitutes a CAD Tag is outside the scope of the JT File format and is indeed part of the CAD system. The JT File format simply provides a way to store any kind of CAD Tag as provided by the CAD system which produced the CAD entity.

Figure 242: Compressed CAD Tag Data collection



I16:Version Number

Version Number is the version identifier for the CADTag element. Only version number 0x001 is currently supported.

I32 : Data Length

Data Length specifies the length in bytes of the Compressed CAD Tag Data collection. A JT file loader/reader may use this information to compute the end position of the Compressed CAD Tag Data within the JT file and thus skip reading the remaining Compressed CAD Tag Data.

I32 : Version Number

Version Number is the version identifier for the Compressed CAD Tag Data 9HUVLRQ QXPEHU ³ ´ LV FXUUHQWO\ WKH RQO\ YDOLG value.

I32 : CAD Tag Count

CAD Tag Count specifies the number of CAD Tags

VecI32{Int32CDP2, Lag1} : CAD Tag Types

CAD Tag Types is a vector of type identifiers for a list of CAD Tags (where each CAD Tag in the list has a type identifier value).

In an uncompressed/decoded form the CAD Tag type identifier values have the following meaning:

= 1	32 Bit Integer CAD Tag Type
= 2	64 Bit Integer CAD Tag Type

CAD Tag Types uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP2, Lag1} : CAD Tags Type-1

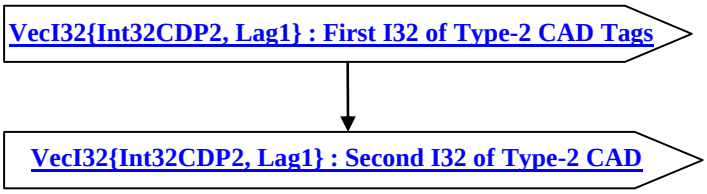
CAD Tags Type-1 is a vector of the Type-1 (i.e. 32 Bit Integer Type) CAD Tags for a list of CAD Tags. CAD Tags Type-1 uses the Int32 version of the CODEC to compress and encode data. CAD Tags Type-1 is only present if there are Type-1 CAD Tags in the CAD Tag Types vector. Thus a loader/reader of JT file must first uncompress/decode and evaluate the previously read CAD Tag Types to determine if there are any Type-1 CAD Tags and if so, then the CAD Tags Type-1 data vector is present.

8.1.16.1 Compressed CAD Tag Type-2 Data

Compressed CAD Tag Type-2 Data collection contains the Type-2 (i.e. 64 Bit integer Type) CAD Tag data for a list of CAD Tags.

The Compressed CAD Tag Type-2 Data collection is only present if there are Type-2 CAD Tags in the CAD Tag Types vector. Thus a loader/reader of JT file must first uncompress/decode and evaluate the previously read CAD Tag Types vector to determine if there are any Type-2 CAD Tags and if so, then the Compressed CAD Tag Type-2 Data collection is present.

Figure 243: Compressed CAD Tag Type-2 Data collection



VecI32{Int32CDP2, Lag1} : First I32 of Type-2 CAD Tags

First I32 of Type-2 CAD Tags is a vector of the first 32 bits of each Type-2 CAD Tag in the list of CAD Tags. First I32 Of Type-2 CAD Tags uses the Int32 version of the CODEC to compress and encode data.

VecI32{Int32CDP2, Lag1} : Second I32 of Type-2 CAD Tags

Second I32 of Type-2 CAD Tags is a vector of the second 32 bits of each Type-2 CAD Tag in the list of CAD Tags. Second I32 Of Type-2 CAD Tags uses the Int32 version of the CODEC to compress and encode data.

8.2 Encoding Algorithms

The following sections give a brief technical overview/descriptions of the various encoding algorithms used in the JT format. Additional information on each of the algorithms can be found within references listed in 3 References and Additional Information section of this document. Also, a sample implementation of the decoding portion of each algorithm can be found in Appendix C: Decoding Algorithms ± An Implementation.

8.2.1 Uniform Data Quantization

Uniform Data Quantization is a lossy encoding algorithm in which a continuous set of input values (floating point data) is approximated with integral multiples (i.e. integers) of a common factor. How close the quantization output approximates the original input data is dependent upon the quantization data range and the number of bits specified to hold the resulting integer value.

7KH TXDQWL]DWLRQ LV FRQVLGHUHG ³XQLIRUP´ EHFDXVH WKH DOJRULWKP GLYLGHV WKH GDWD LQSXW UDQJH LQWR 0HYHQ RI HTXDO VSDFLQJ L H. a uniform scale). The form of Uniform Data Quantization used by the JT format is also considered scalar in nature, in that each input value is treated separately in producing the output integer value.

Given the following definitions:

```
inputVal:      Input floating point data to quantize
outputval:     Resulting quantized output integer value
minInputRange: Specified minimum value of input data range
maxInputRange: Specified maximum value of input data  range
nBits:         Specified number of bits of precision (quantized size)
```

The basic algorithm (using C++ style syntax) for Uniform Data Quantization is as follows:

```
UInt32 iMaxCode = (nBits < 32) ? (0x1 << nBits) - 1 : 0xffffffff;
Float64 encodeMultiplier = Float64(iMaxCode) / (maxInputRange – minInputRange);
UInt32 outputVal = UInt32( (inputVal - minInputRange) * encodeMultiplier + 0.5 );
```

1RWH)RU UHDVRQV RI UREXVWQHVV ³RXWSXW9DO´ PXVW DOVR EH H[SOLFLW\ FODPSHG WR WKe range [0,iMaxCode]. This is because floating-SRLQW URXQGRII HUURU LQ WKH FDOFXODWLRQ RI ³HQFRGH0 XOWLSOLHU´ FDQ RWKHUZLVH FDXVH ³RXWSXW9DO´ WR VRPHWLPHV FRPH RXW HTXDO WR ³L0 D[&RGH ´

Note that all compression algorithms in the following sections operate on quantized integer data.

8.2.2 Bitlength CODEC

This is a very simple compression algorithm that runs an adaptive-width bit field encoding for each value. As each input value is encountered, the number of bits needed to represent it is calculated and compared to the current "field width". The FXUUHQW ILHG ZLGWK LV WKHQ DGMXVWHG XSZDUGV RU GRZQZDUGV E\ D FRQVWDQW ³VHSBVL]H´ QXPEHU RI ELWV L H ELWV IRU WKH -7 format) to accommodate the input value storage. This increment or decrement of the current field width is indicated for each encoded value by a prefix code stored with each value.

The prefix code will be one of the following two forms:

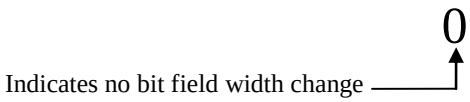
A single '0' bit to denote the same (i.e. current) field width is to be used for the next value.

A '1' bit followed by a series of one or more bits where each bit indicates whether the field width is to be incremented (a '1' bit) or decremented (a '0' bit) by the field step_size, followed by a single terminator bit (which is complement of the previous increment/decrement bit). Note that there can only be increments or decrements in a given prefix code, never both, and that

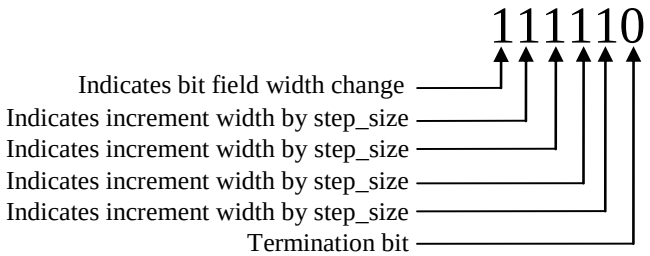
is why the prefix code terminator bit can be recognized as bits are read by simply looking for the complement of the previous increment/decrement bit.

Some examples of prefix codes and their interpretation are as follows:

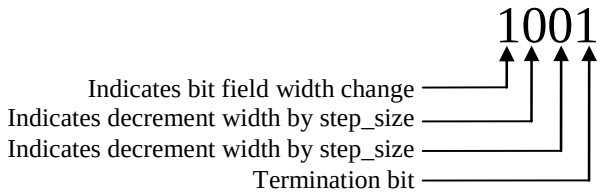
Example 1: Prefix code to maintain same (current) field width.



Example 2: Prefix code to increment field width four times (8 bits).



Example 3: Prefix code to decrement field width two times.



A pseudo-code sample implementation of bit length decoding is available in Appendix C: Decoding Algorithms ± An Implementation.

8.2.3 Arithmetic CODEC

,Q &0DXGH 6KDQQRQ RI %H00 /DERUDWRULHV SXE0LVKHG KLV VHPLQDO SDSHU ³$ PDWKHPDWLFDO WKHRU\ RI FRPPXQLFDWLRQ´ that launched the new field of Information Theory. In that same year, two Doctoral students at the Massachusetts Institute of Technology (MIT) made breakthroughs in the coding of information. The first to press was David Huffman, whose coding scheme we now know as Huffman Coding. In that same class with Huffman was Peter Elias who reportedly developed the first articulation of arithmetic coding, but it lay unpublished until 1976, when Jorma Rissanen and Richard Pasco, of IBM, refined it into a practically useful algorithm.

Arithmetic encoding is a lossless compression algorithm that replaces an input stream of symbols or bytes with a single fixed point output number (i.e. only the mantissa bits to the right of the binary point are output from MSB to LSB). The total number of bits needed in the output number is dependent upon the length/complexity of the input message (i.e. the longer the input message the more bits needed in the output number). This single fixed point number output from an arithmetic encoding process must be uniquely decodable to create the exact stream of input symbols that were used to create it.

Initially all symbols being encoded have a probability value assigned to them based on the likelihood that the symbol will occur next in the input stream (i.e. the frequency of the symbol in the input stream). Given probability value assignments, HDFK LQGLYLGXD0 V\PER0 LV WKHQ DVVLJQHG DQ LQWHUYD0 UDQJH D0RQJ D QRPLQD0 WR ³SUREDELOLW\ 0LQH´ ZKHUH WKH VL]H RI HDFK range corresponds to the s\PER0¶V SUREDELOLW\ YD0XH 1RWH WKDW D SDUWLFX0DU V\PER0 RZQV D00 YD0XHV ZLWKLQ LWV DVVLJQHG UDQJH up to, but not including, the range high value, and that it does not matter which symbols are assigned which segment of the range as long it is done in the same manner by both the encoder and the decoder.

Given the above described input stream probability and interval range assignments, a high level description of the arithmetic encoding process is as follows:

%HJLQ ZLWK D ³FXUUHQW LQWHUYD0´ LQLWLD0L]HG WR > 1RWH WKDW LQ LQWHUYD0 UDQJH QRWDWLRQ L H ³> ´ WKH ³>³ V\PER0 LQGLFDWHV LQF0XVLYH RI WKH LQWHUYD0 0RZ 0LPLW DQG ³ ´ V\PER0 LQGLFDWHV H[F0XVLYH RI WKH LQWHUYD0 KLJK 0LPLW

Sequentially for each symbol of the input stream, perform two steps

Subdivide the current interval into subintervals based on the input stream symbol probability values as described above.

Select the subinterval corresponding to the current input stream symbol being sequentially processed and make it the new ³FXUUHQW LQWHUYD0´

$IWHU D00 LQSXW VWUHDP V\PER0V KDYH EHHQ VHTXHQWLD00\ SURFHVVHG RXWSXW HQRXJK ELWV WR GLVWLQJXLVK WKH ILQD0 ³FXUUHQW LQWHUYD0´ from all other possible final intervals.

In pseudo code form, the algorithm to accomplish the above described arithmetic encoding for an input stream message of any length could look as follows:

```
Set low to 0.0
Set high to 1.0
While there are still input symbols do
    cur_symbol = get next input symbol
    range = high – low
    high = low + range * high_range(cur_symbol)
    low = low + range * low_range(cur_symbol)
End of While
Output low
```

So the arithmetic encoding process is simply one in which we narrow the range of possible numbers with every new sequentially processed input symbol; where the new narrowed range is proportional to the predefined probability values assigned to each symbol in the input stream.

The arithmetic decoding process is the inverse procedure; where the range is expanded in proportion to the probability of each symbol as it is extracted. For the arithmetic decoding process we find the first symbol in the message by seeing which symbol owns the interval range that our encoded message falls in. Then, since we know the low and high range limit values of the first symbol we can remove their effects by reversing the process that put them in.

In pseudo code form, the algorithm for decoding the incoming number could look as follows:

```
Get encoded_number
Do
    find symbol whose range straddles the encoded_number
    output the symbol
    range = symbol_high_value – symbol_low_value
    encoded_number = encoded_number – symbol_low_value
    encoded_number = encoded_number / range
until no more symbols
```

8.2.3.1 Example

Following is an example to demonstrate in practice the basic principles of arithmetic coding.

Suppose you want to compress, using arithmetic coding, the following sequence/array of integer data:

{2, 9, 12, 12, 0, 7, 1, 20, 5, 19}

For this input stream of data, the assigned probability values will be as follows:

Number	Probability
0	1/10
1	1/10
2	1/10
5	1/10

Number	Probability
7	1/10
9	1/10
12	2/10
19	1/10
20	1/10

7KHQ EDVHG RQ HDFK LQSXW QXPEHUV SUREDELOLW\ YDOXH DQ LQWHUYDO UDQJH DORQJ D WR ³SUREDELOLW\ 0LQ´ FDQ EH DVVLJQHG WR each input number as follows:

Number	Probability	Range
0	1/10	[0.00, 0.10)
1	1/10	[0.10, 0.20)
2	1/10	[0.20, 0.30)
5	1/10	[0.30, 0.40)
7	1/10	[0.40, 0.50)
9	1/10	[0.50, 0.60)
12	2/10	[0.60, 0.80)
19	1/10	[0.80, 0.90)
20	1/10	[0.90, 1.00)

Now proceeding with encoding the example input integer sequence {2, 9, 12, 12, 0, 7, 1, 20, 5, 19}, the first number to be HQFRGHG LV ³ ´ VR WKH ILQDO HQFRGHG YDOXH ZLOO EH D QXPEHU WKDW LV JUHDWHU WKDQ RU HTXDO WR DQG 0HVV WKDQ 1RZ DV each subsequent number in the input stream is sequentially processed for encoding, the possible range of the output number is IXUWKHU UHVWULFWHG ,Q RXU H[DPS0H WKH QH[W QXPEHU WR EH HQFRGHG LV ³ ´ ZKLFK RZQV WKH UDQJH > ZLWKLQ WKH QHZ sub-range of [0.20, 0.30); which now further restricts our output number to the range [0.25, 0.26). If we continue this logic for the complete input integer sequence we end up with the following:

New integer number	Low value	High value
	0.0	1.0
2	0.2	0.3
9	0.25	0.26
12	0.256	0.258
12	0.2572	0.2576
0	0.25720	0.25724
7	0.257216	0.257220
1	0.2572164	0.2572168
20	0.25721676	0.2572168
5	0.257216772	0.257216776
19	**0.2572167752**	0.2572167756

)URP WKH DERYH WDE0H DUH ILQDO 0RZ YD0XHV LV ³ ´ ZKLFK LV WKH RXWSXW QXPEHU WKDW XQLTXH0\ HQFRGHV WKH LQWHJHU number sequence {2, 9, 12, 12, 0, 7, 1, 20, 5, 19}.

Given this encoding scheme, the decoding would simply follow the process previously described. We find the first number in WKH VHTXHQFH E\ 0RRNLQJ XS LQ WKH SUREDELOLW\ UDQJH IRU WKH YD0XH ZKRVH UDQJH RXU HQFRGHG QXPEHU ³ ´ ID00V

within. In ouU H[DPSOH WKLV HTXDWHV WR WKH YDOXH ³ ´ DQG VR RXU ILUVW GHFRG K

	High	Low	Range	Cumulative output
Shift out 2	39999	00000	40000	.2572
(QFRGH ³ ´ >	19999	16000		.2572
Shift out 1	99999	60000	40000	.25721
(QFRGH ³ ´ >	67999	64000		.25721
Shift out 6	79999	40000	40000	.257216
(QFRGH ³ ´ >	79999	76000		.257216
Shift out 7	99999	60000	40000	.2572167
(QFRGH ³ ´ >	75999	72000		.2572167
Shift out 7	59999	20000	40000	.25721677
(QFRGH ³ ´ >	55999	52000		.25721677
Shift out 5	59999	20000	40000	.257216775
Shift out 2				.2572167752
Shift out 0				.25721677520

As can be seen in the above table, after all values in the input stream have been encoded and any final matching most significant digit has been output, the arithmetic coding algorithm requires that two extra digits be shifted out of either the high or low value to finish up the cumulative output word.

Although the above example incrementally encodes very nicely with the arithmetic coding algorithm, there are certain cases where the computed high and low values get closer, but never actually converge to one value in the most significant digit (e.g. High = 0.300001, Low = 0.29992). Thus after a few iterations the difference between high and low becomes so small that 16 bits is not sufficient to represent any difference between the values (i.e. all calculations return the same values). This FRQGLWLRQV LV NQRZQ DV ³XQGHUIORZ´ DQG VSHFLDO ORJLF PXVW DGGHG WR WKH DULWKPHWLF FRGLQJ DJRULWKP WR UHFRJQL]H WKDW ³XQGHUIORZ´ LV RFFXUULQJ DQG WKXV KHDG LW RII EHIRUH WKH Fomputations reach an impasse.

7KH DGGLWLRQDO ORJLF IRU UHFRJQL]LQJ WKDW ³XQGHUIORZ´ LV RFFXUULQJ ZRX0G EH H[HFXWHG DIWHU HDFK UHFD0FX0DWLRQ RI +LJK DQG Low value set, and in pseudo code form this logic would look as follows:

underflow = FALSE

if((High DQG /RZ YD0XH¶V VLJQLILFDQW GLJLWV GRQ¶W PDWFK EXW DUH RQ DGMDFHQW QXPEHUV

(2nd PRVW VLJQLILFDQW GLJLW RI +LJK LV ³ ´ DQG WKH nd PRVW VLJQLILFDQW GLJLW RI ORZ LV ³ ´

{

underflow = TRUE

}

: KHQ ,I LW LV LGHQWLILHG WKDW ³XQGHUIORZ´ LV RFFXUULQJ WKH encoding algorithm must perform the following steps to stop the FXUUHQW ³XQGHUIORZ´

Delete the 2nd most significant digit from both the High and Low value.

Shift the other digits (those to the right of the deleted 2nd digit) to the left to fill up the space (note that the most significant digit stays in place).

,QFUHPHQW D FRXQWHU WR UHPHPEHU WKDW ZH WKUHZ DZD\ D GLJLW DQG GRQ¶W NQRZ ZKHWKHU LW ZDV JRLQJ WR FRQYHUJH WR ³ ´ RU ³ ´

A before and after example of performing the above steps to the High and LRZ YD0XHV ZKHQ µXQGHUIORZ´ RFFXUV LV DV follows:

	Before	After
High	40344	43449
Low	39810	38100
Underflow_counter	0	1

Now as the encoding algorithm continues and the most significant digit of High and Low values once again converge to a common vDOXH WKHQ WKDW YDOXH PXVW EH RXWSXW WR WKH FRGHG ZRUG DORQJ ZLWK ³8QGHUIORZBFRXQWHU´ QXPEHU RI ³XQGHUIORZ´ digits that were previously deleted. The underflow digits output to the coded word will either be all 9s or 0s, depending on whether the High and Low value converged to the higher or lower value.

A pseudo-code sample implementation of arithmetic decoding is available in Appendix C: Decoding Algorithms ± An Implementation.

8.2.4 Deering Normal CODEC

Michael Deering first published his work on geometry compression in 1995 [5] and later helped present a course on the VXEMHFW DW 6,**5$3+¶ [6] $OWKRXJK 'HHULQJ¶V DSSURDFK WR JHRPHWULF FRPSUHVVLRQ LQYROYHV FRPSUHVVLRQ RI YHUWLFHV colors and normals, the description detailed here will focus solely on compression of normals since this is the only FRPSRQHQW RI 'HHULQJ¶V DSSURDFK XVHd in the JT format.

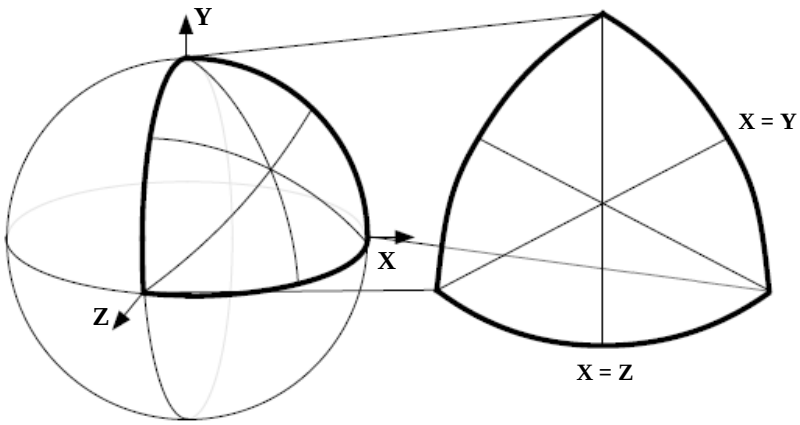
Through both theoretical examination and empirical testing, Deering found that an angular density of 0.01 radians between normals (about 100,000 normalized normals distributed over unit sphere) gave results that were not visually distinguishable IURP UHVXOWV REWDLQHG IURP ILQHU QRUPDO UHSUHVHQWDWLRQV 7KLV REVHUYDWLRQ UHGXFHG WKH SUREOHP RI KDYLQJ WR ³H[DFWO\´ represent any general surface normal, to only having to represent about 100,000 specific normals (i.e. general surface normal replaced by the appropriate one of the 100,000 specific normals).

If there were no run-time memory concerns and no concerns for on disk footprint size, these specific 100,000 normals could be simply represented in a table that is indexed into, to reference D SDUWLFXODU QRUPDO ,QVWHDG 'HHULQJ¶V DSSURDFK OHYHUDJHV symmetrical properties of the unit sphere to reduce the size of the table and allow any normal to be represented by, at max, an 18 bit index as summarized below:

- All normals are normalized (i.e. can be represented as points on the surface of the unit sphere).
- 8QLW VSKHUH LV GLYLGHG LQWR HLJKW V\PPHWULFDO RFWDQWV EDVHG RQ VLJQ ELWV RI QRUPDO¶V ; < = UHFWLOLQHDU representation (see Figure 244). Using three bits to represent the three sign bits of the normals XYZ components reduces the problem space to one eighth of the unit sphere
- Each octant of the unit sphere is divided into six identical sextants by folding about the planes of symmetry; x=y, x=z, and y=z (see Figure 244). The particular sextant can be encoded using another three bits. So now unit sphere is divided into 48 identically shaped triangle patches reducing the normal look-up table to about 2000 entries (i.e. 100000/48).
- Then, a local rectangular orthogonal two dimensional grid is created on the sextant and all normals within the sextant are represented as two n-bit angular addresses (i.e. a quantization of two angular values along the unit VSKHUH ZKHUH ³Q´ LV LQ WKH UDQJH IURP WR ELWV
- Resulting in a max grand total of 18 bits (3 + 3 + 6 + 6) to represent any normal on the unit sphere.

In the figure below, the sphere is divided into eight octants and each octant is divided into six sextants. Each sextant is assigned an identifying three bit code.

Figure 244: Sextant Coding on the Sphere



Note that the sextant three bit code assignments used by the JT format (as seen in Figure 244) are slightly modified from the original assignments as specified by Deering.

The representation of all normals within a sextant by two n-bit angular addresses, as summarized above, is based on the following:

- In spherical coordinates, points on a unit sphere can be parameterL]HG E\ WZR DQJ0HV θ DQG φ ZKHUH θ LV WKH DQJ0H DERXW WKH \ D[LV DQG φ LV WKH 0RQJLWXGLQD DQJ0H IURP WKH \ S0DQH
- Mapping between rectangular and spherical coordinates is:
  [FRVθ FRVφ \ VLQφ] VLQθ FRVφ
- All encoding takes place in the positive octant.
- $QJ0HV θ DQG φ FDQ EH TXDQWL]HG LQWR WZR Q-ELW LQWHJHUV θ¶n DQG φ¶n ZKHUH ³Q´ LV LQ WKH UDQJH RI WR DQG WKH relationship between these n-ELW LQWHJHUV DQG DQJ0HV θ DQG φ IRU D JLYHQ ³Q´ LV
  θ θ¶n) = asin WDQ φmax * (n ± θ¶n) / 2n)
  φ φ¶n φmax φ¶n / 2n

Thus to encode (i.e. quantize) a given normal **N** LQWR θ¶n and φ¶n:

- N must be first represented (see Figure 244) in the positive octant and appropriate sextant within that octant, UHVXOWLQJ LQ 1¶
- 7KHQ 1¶ PXVW EH GRWWHG ZLWK D0 TXDQWL]HG QRUPDOV LQ WKH VH[WDQW
-)RU D IL[HG ³Q´ WKH FRUUHVSRQGLQJ θ¶Q DQG φ¶Q YD0XHV RI WKH TXDQWL]HG VH[WDQW QRUPD0 WKDW UHVXOW LQ WKH 0DUJHVW QHDUHVW XQLW\ GRW SURGXFW GHIILQHV WKH SURSHU θ¶Q DQG φ¶Q HQFRGLQJ RI 1

With this encoding of normal **N** LQWR θ¶n and φ¶n n-bit integers the complete bit representation of normal **N** can now be defined as follows:

- Uppermost three bits specify the octant.
- Next three bits specify the sextant code as defined in Figure 244.
- Next two n-bLW ILH0GV VSHFL\ θ¶Q DQG φ¶Q YD0XHV UHVSHFWLYH0\

8.3 ZLIB Compression

ZLIB compression is a lossless data compression algorithm and LV HVVHQWLD0\ WKH VDPH DV WKDW LQ J]LS DQG =LS =0LE¶V compression method, called deflation, creates compressed data as a sequence of blocks. The JT format uses *Version 1.1.2* of the ZLIB compression library.

9 Best Practices

The proceeding sections of this document specify the mandatory clauses for creating a reference compliant Version 9.5 JT file. 7KLV ³Best Practices´ VHFWLRQ focusing on documenting format conventions that although not required to have a

reference compliant JT file, have become commonplace within JT format translators to the point where these conventions are considered *best practices* for constructing JT files.

9.1 Late-Loading Data

The JT format was designed and structured to load entities from a JT file on a deferred or as-needed basis.. This concept is UHIHUUHG WR ZLWKLQ WKLV -7)RUPDW 5HIHUHQFH GRFXPHQW DV ³0DWH-0RDGLQJ GDWD´ 7KH -7 IRUPDW KDV PDQ\ VWUXFWXUHV LQ VXSSRUW of this and it is recommended as a best practice that writers/loaders of JT data leverage these capabilities.

Initial loading only requires the Table of Contents and the LSG. All Meta Data Node Elements, JT B-Rep Elements, XT B-Rep Elements, Wireframe Rep Elements, PMI Manager Meta Data Elements, JT ULP Elements, JT LWPA Elements, and Shape LOD Elements may be ignored until they are actually needed. These Late-Loaded data containers are accessed via a Late Loaded Property Atom Element which appears in a LSG Node's Property list. Contained in this Property is the GUID associated with the segment to be loaded. This GUID can be looked up in the TOC Segment, which will give the location in the JT from which to load the actual Element via the Data Segment convention.

9.2 Bit Fields

In the 7 File Format section of this reference many bit field data descriptions (e.g. 7.2.1.1.1.1.1 Base Node Data ³1RGH)ODJV´ ILHOG FRQWDLQ WKH ZRUGV ³*All undocumented bits are reserved*´ These words should be interpreted to mean that these undocumented bits should EH VHW WR ³´ ZKHQ ZULWLQJ WKH ELW ILHOG GDWD WR D -7 ILOH

9.3 Reserved Field

In the 7 File Format section of this reference VRPH GDWD ILHOGV PD\ EH QDPHG GRFXPHQWHG ³5HVHUYHG)LHOG´ H J 7.2.1.1.1.7.1LOD Node Data ´5HVHUYHG)LHOG´ ILHOG $ ³5eVHUYHG)LHOG´ H[LVWV IRU SRWHQWLDO IXWXUH H[SDQVLRQ RI WKH)RUPDW and best practices suggests that these fields should be treated as follows:

If you are writing a JT file whose data did not originate from reading a previous JT file, then Reserved Fields should be set to D YDOXH D ³´ ZKHQ ZULWLQJ WKH ILHOG WR D -7 ILOH

If you are writing a JT file whose data originated from reading a previous JT file (i.e. rewriting D -7)LOH WKHQ ³5HVHUYHG)LHOGV´ VKRXOG EH ZULWWHQ ZLWK the same value that was read from the originating JT file.

9.4 Local Version

The local version values seen throughout the data collections provides a simple means by which those data collections can be extended within current and future minor versions of the 9.x file format. The standard convention followed by each data collection, unless explicitly specified otherwise, is to write the data from each local version in order. This allows readers to read up to the maximum local version they support and then use the segment length that was read in the Segment Header to skip over any data they may not understand.

9.5 Hash Value

Hashing is a means by which a large chunk of values can be represented by single value through the use of a mathematical function that provides a distinctive value for each unique set of ordered values. The hash function used within the v9.x format was published by Bob Jenkins in Dr Dobbs back in 1997 and its implementation is provided in Appendix D: .

The hash function takes a pointer to a set of values, the number of values, and a seed hash value. It returns the resulting hash value. Initially the seed value is set to 0, however when hashing multiple data fields together the hash of previous data field is used as the seed hash value of the next data field:

```
UInt32 uHash = 0;
uHash = hash32( pVal0, nVal0, uHash );
uHash = hash32( pVal1, nVal1, uHash );
```

The order that individual fields are hashed is extremely important since v9.x readers are strongly encouraged to assert that the stored hash value matches the calculated hash value of the corresponding fields after reading in all the corresponding data. To this end each hash value stored within the v9.x format carefully documents which fields it encompasses and the order in which they should be hashed.

9.6 Metadata Conventions

Although there are really no restrictions/limits/requirements on what metadata (i.e. properties) can/must be attached to nodes in the LSG in order to have a reference compliant JT file, there are some conventions that have been generally followed in the industry when translating CAD data to the JT file format. See 7.2.1.2 Property Atom Elements section of this document for complete description of the file Elements used to attach this property information to nodes.

9.6.1 CAD Properties

The following table lists the conventions that CAD data translators typically (although not always) follow in placing CAD information in a JT file as properties on various LSG nodes. Some of these properties are considered required in order for the data in the file to be interpreted correctly while other properties are optional. See flowing sub-sections for additional information on required versus optional properties.

The convention is to place these Units properties on every Part and Assembly grouping node in the LSG. By following this convention, JT file format readers/writers are provided maximum flexibility in understanding/indicating the appropriate JT data unit processing for both, monolithic and shattered JT file Assembly structures.

JT Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values	Required / Optional
JT_PROP_MEASUREMENT_UNITS	Model Units	MbString	MbString	millimeters centimeters meters inches feet yards micrometers decimeters kilometers mils miles	Required
CAD_MASS_UNITS	Units of mass	MbString	MbString	micrograms milligrams grams kilograms ounces pounds	Required
CAD_SURFACE_AREA	Surface area of solids within part.	MbString	F64	numeric	Optional
CAD_VOLUME	Volume of solids within part	MbString	F64	numeric	Optional
CAD_DENSITY	Density of solids within part (6)	MbString	F64	numeric	Optional
CAD_MASS	Mass or weight of solids within part	MbString	F64	numeric	Optional
CAD_CENTER_OF_GRAVITY	Center of gravity of solids within part	MbString	3 space separated F64	3 numeric values	Optional
CAD_PROP_MATERIAL_THICKNESS	Sheet thickness within part	MbString	F64	numeric	Optional
CAD_PART_NAME	Component name from translator	MbString	MbString	<string>	Optional
CAD_SOURCE	CAD program the Part originated from	MbString	MbString	<string>	Optional

Table 9: CAD Property Conventions

9.6.1.1 Required Properties

The required unit properties are really necessary for viewers of JT file data to properly interpret numeric data for analysis operations (e.g. measure) and support the building of assemblies through the reading of multiple JT files in disparate units. There are two units of measure that are relevant, units of distance and units of weight.

The JT_PROP_MEASURMENT_UNITS property is used to specify units of distance. The CAD_MASS_UNITS property is used to specify units for weight. JT_PROP_MEASURMENT_UNITS property is strictly required, while CAD_MASS_UNITS property is "optionally required". %\ ³RSWLRQDO\ UHTXLUHG´ ZH PHDQ LW LV UHTXLUHG LI RWKHU RSWLRQDO metadata intends to specify properties that would depend on these units of measure (e.g. CAD_DENSITY and CAD_MASS). Notice that the Mass units are specified, instead of the Density units, since Density is a derived unit of Mass/Volume.

9.6.1.2 Optional Properties

Optional properties can be provided, but if the property is a units based value, then the value must be in units that are

JT Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values
	³ ´			
Angular::	Angular tessellation tolerance for each LOD in degrees. Two consecutive segments in a linear approximation of a curve/surface form an angle; this value specifies the maximum angle allowed. Encoded value string would look as follows for the case of two LODs: ³ ´	MbString	space separated F32 values	Numeric

9.6.3 Miscellaneous Properties

The below table documents some miscellaneous properties often placed on various nodes in the LSG to communicate specific information about the node or its contents.

JT Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values
PMI_TYPE_TABLE	May be attached to Part Node Element to indicate the list of PMI type values and associated names for all PMI types (basically equivalent to the Entity Type field documented in Generic PMI Entities). The string is a ³ ´ DQG ³ ´ delimited string of the following form: ³10.Groove Weld,11.Fillet Weld,12.Plug/Slot Weld,14.Edge Weld´	MbString	<string>	
JT_PROP_SHAPE_DATA_TYPE	May be attached to Shape Node Elements to indicate what geometry type the shape data represents.	MbString	<string>	³6XUIDFH´ ³ : LUH´
~~JT_PROP_TRISTRIP_DATA_LAYOUT~~	This property is deprecated, and is no longer used.			
JT_PROP_ORIGINATING_BREPTYPE	May be attached to Part Node Element to indicate the type of B-Rep associated with the Part.	MbString	<string>	³1RQH´ ³-W%UHS´ ³;7%UHS´
JT_PROP_NAME	May be attached to any form of node or attribute with which one wants to associate a textual name (e.g. Part/Assembly/Instance name, Material name, Light Set name, etc.). For Part/Assembly/Instance names this string has the following encoded form ZKHUH ³ ´ LV D GHOLPLWHU DQG ³ ¶ is a terminator:	MbString	<string>	

JT Property Key	Meaning	JT File Data Type	Encoded Data Type	Valid Values
	"**Alignment**Pin.part;0;1:" Name Version # Instance # For attribute names this string has the following encoded form: "**Chrome material**" Name			

9.7 LSG Attribute Accumulation Semantics

For applications producing or consuming JT format data, it is important that the JT format semantics of how attributes are meant to be applied and accumulated down the LSG are followed. If not followed, then consistency between the applications in terms of 3D positioning and rendering of LSG model data will not be achieved.

Although each attribute type defines its own application and accumulation LSG semantics (the details of which can be found in each attribute type sub-section under 7.2.1.1.2 Attribute Elements), there are some general rules which apply:

Attributes at lower level in the LSG take precedence and replace or accumulate with attributes set at higher levels. When multiple Attributes of the same type are present on a Node, they accumulate in the order they are specified (i.e. from the front of the Attribute list toward the back).

Nodes with no associated attributes inherit those of their parents.

Attributes are inherited only IURP D QRGH V SDUHQWV 7KXV D JLYHQ QRGH¶V DWWULEXWHV GR QRW DIIHFW WKRVH RQ WKH QRGH V VLEOLQJV

The root of a partition inherits the attributes in effect at the referring partition node.

$WWULEXWHV FDQ EH PDUNHG ³ILQDO´ ZKLFK WHUPLQDWHV DFFXPXODWLRQ RI WKDW DWWULEXWH W\SH DW WKDW PDUNHG DWWULEXWH DQG SURSDJDWHs the accumulated value at that point to all descendants of the associated node. Descendants can override a "final´ DWWULEXWH XVLQJ WKH ³IRUFH´ IODJ 1RWH WKDW ³IRUFH´ GRHV QRW WXUQ 2)) ³ILQDO´ ± it is simply a one-VKRW RYHUULGH RI ³ILQDO´ IRU WKH VSHFLILF DWWULEXWH PDUNHG DV ³IRUFLQJ´ 0 XOWLSOH DWWULEutes of the same type may be marked as "forcing" and in this case, the 0DVW RQH ZLQV %RWK RI WKHVH IODJV DUH 2)) E\ GHIDXOW $Q DQDORJ\ IRU WKLV ³IRUFH´ DQG ³ILQDO´ LQWHUDFWLRQ LV WKDW ³ILQDO´ is a back-door in the attribute accumulation semantics, and WKDW ³IRUFH´ LV WKH GRJJ\-door in the back-door!

9.8 LSG Part Structure

The JT Format Reference does not mandate that a particular node hierarchy be used for modeling physical Parts within a LSG structure. In fact there are many node hierarchies for representing Parts in LSG that will function correctly in most JT enabled applications. Still, there is a convention that most JT translators follow (and some JT enabled applications may assume exists) for modeling Parts within a LSG. The convention is to model each Part within a LSG structure with the following node hierarchy:

Figure 245: JT Format Convention for Modeling each Part in LSG

9.9 Range LOD Node Alternative Rep Selection

Best practices suggest that LSG traversers apply the following strategy, at Range LOD Nodes (see 7.2.1.1.1.8 Range LOD Node Element), when making alternative representation selection decisions based on Range Limits: The first alternate representation is valid when the world coordinate distance between the center and the eye point is less than or equal to the first range limit (and when no range limits are specified). The second alternate representation is valid when the distance is greater than the first limit and less than or equal to the second limit, and so on. The last alternate representation is valid for all distances greater than the last specified limit.

9.10 Brep Face Group Associations

The original purpose of the face group concept was to provide associativity between Brep faces and geometry. Exactly how a Brep face associates to a face group number is the topic of this section. An implicit scheme has been chosen for face group associativity, rather than storing some kind of explicit data on either the Vertex Shape LOD Data or the Brep. The primary motivation for this implicit scheme is to keep the JT files simple and small; additional association information would not only be redundant, but also wasteful. Tessellators must exercise this policy when producing Vertex Shape LOD Data from Breps, grouping the triangles into face groups according to its rules. Tristrips may not cross face groups. Applications must be able to count on this policy so that, for example, they can map a picking action back to its corresponding Brep face reliably.

JTBrep/ULP: In the case of JtBrep and ULP reps, the mapping is simple. These Reps have a consistent, sequential, index origin-0 numbering scheme for their regions, shells, and faces. So the Brep faces are simply assigned sequentially to face group by increasing region and shell. For example, suppose we have a JTBrep with 2 regions, each with 2 shells, each with 2 faces. The Face Group ⇔ Region/Shell/Face mapping will be as follows: